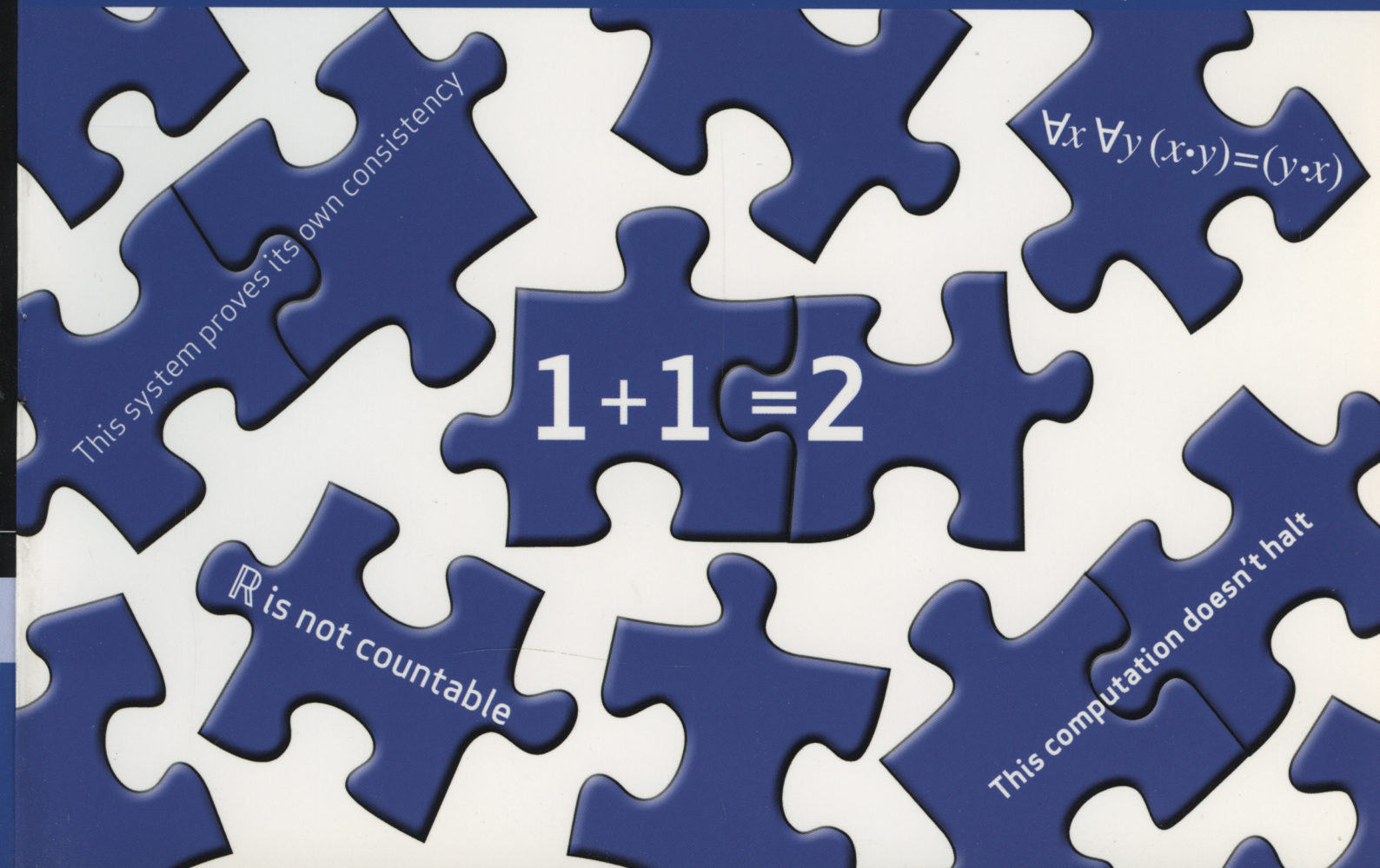
The Open University

M381 Number Theory and Mathematical Logic

Mathematical Logic **Unit 3**

Church's Thesis



The Open University

M381 Number Theory and Mathematical Logic

Mathematical Logic **Unit 3**

Church's Thesis

Prepared by the Course Team

The M381 Mathematical Logic Course Team

The Mathematical Logic half of the course was produced by the following team:

Roberta Cheriyan	*Course Manager*
Derek Goldrei	*Course Team Chair* and *Academic Editor*
Jeremy Gray	*History Consultant*
Mary Jones	*Critical Reader*
Roger Lowry	*Publishing Editor*
Alan Pears	*Author*
Alan Slomson	*Author*
Frances Williams	*Critical Reader*

with valuable assistance from:

The Maths Production Unit, Learning & Teaching Solutions:
Becky Browne, Jim Campbell, Nicky Kempton, Bill Norman, Sharon Powell, Katie Sayce, Penny Tee

Alison Cadle	*TEX Consultant*
Michael Goldrei	*Cover Design Consultant*
Vicki McCulloch	*Cover Designer*

The external assessor was:

Jeff Paris	*Professor of Pure Mathematics, University of Manchester*

The Course Team would like to acknowledge their reliance on the previous work of Alan Slomson and of Alex Wilkie, Professor of Mathematical Logic, University of Oxford. We would also like to thank the family of Alonzo Church for generously providing the photo of him used in this unit.

This publication forms part of an Open University course. Details of this and other Open University courses can be obtained from the Student Registration and Enquiry Service, The Open University, PO Box 197, Milton Keynes, MK7 6BJ, United Kingdom: tel. +44 (0)870 300 6090, e-mail general-enquiries@open.ac.uk

Alternatively, you may visit the Open University website at http://www.open.ac.uk where you can learn more about the wide range of courses and packs offered at all levels by The Open University.

To purchase a selection of Open University course materials, visit http://www.ouw.co.uk, or contact Open University Worldwide, Michael Young Building, Walton Hall, Milton Keynes, MK7 6AA, United Kingdom, for a brochure: tel. +44 (0)1908 858793, fax +44 (0)1908 858787, e-mail ouw-customer-services@open.ac.uk

The Open University, Walton Hall, Milton Keynes, MK7 6AA.

First published 2004. Reprinted as new edition 2007, with corrections.

Copyright © 2004 The Open University

All rights reserved; no part of this publication may be reproduced, stored in a retrieval system, transmitted or utilised in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without written permission from the publisher or a licence from the Copyright Licensing Agency Ltd. Details of such licences (for reprographic reproduction) may be obtained from the Copyright Licensing Agency Ltd, Saffron House, 6–10 Kirby Street, London EC1N 8TS; website http://www.cla.co.uk.

Open University course materials may also be made available in electronic formats for use by students of the University. All rights, including copyright and related rights and database rights, in electronic course materials and their contents are owned by or licensed to The Open University, or otherwise used by The Open University as permitted by applicable law.

In using electronic course materials and their contents you agree that your use will be solely for the purposes of following an Open University course of study or otherwise as licensed by The Open University or its assigns.

Except as permitted above you undertake not to copy, store in any medium (including electronic storage or use in a website), distribute, transmit or re-transmit, broadcast, modify or show in public such electronic materials in whole or in part without the prior written consent of The Open University or in accordance with the Copyright, Designs and Patents Act 1988.

Edited, designed and typeset by The Open University, using the Open University TEX System.

Printed in the United Kingdom by Charlesworth Press, Wakefield.

ISBN 978 0 7492 2266 6

3.1

CONTENTS

Introduction 4

1 URM-computable Functions 4

1.1 A diagonal argument 5

1.2 Partial functions and URM-computability 6

2 Recursive Functions 8

2.1 Unbounded searches 8

2.2 Coding URM computations 19

2.3 URM-computable functions are recursive 24

3 Scope and Limits of Computability 29

3.1 Church's Thesis 29

3.2 Algorithmically undecidable problems 34

Summary 40

Objectives 41

Additional Exercises 41

Solutions to the Problems 45

Solutions to Additional Exercises 51

Index 59

INTRODUCTION

In this unit we complete our study of computable functions and derive some results about their scope and limitations.

In *Unit 1* we studied URM-computable functions directly in terms of URM programs, but in *Unit 2* we changed our approach and looked instead at two processes for generating URM-computable functions, namely substitution and primitive recursion. The functions that we can obtain from a particular set of basic functions using these processes are called the primitive recursive functions. From the work in *Unit 1* we were able to deduce immediately that all these primitive recursive functions are URM-computable. In *Unit 2*, we explored the class of primitive recursive functions in some detail and showed that very many useful functions are primitive recursive.

Despite the power of primitive recursive functions, we were able to prove, using Cantor's ideas, that not all functions are primitive recursive. This followed from the observation that the set of primitive recursive functions of one variable is countably infinite while the set of all functions of one variable is infinite but not countably infinite. Since we showed that URM-computable functions also form a countably infinite set, this sort of argument is not adequate to answer the question as to whether every URM-computable function is primitive recursive, but we have hinted that this is not the case. We explore this point further in this unit.

We begin, in Section 1, with a diagonal argument to show that there is a URM-computable function which is not primitive recursive. We do not give all the details but we hope that you will find the argument convincing. Then we broaden our notion of URM-computability in a way that we shall see is important. It will be clear that some functions in this wider class of URM-computable functions are not primitive recursive. This leads, in Section 2, to the introduction of a third process, *minimization*, for generating functions. We show that, together with substitution and primitive recursion, this process generates all the URM-computable functions in the new wider sense. In Section 3, we use this to deduce a number of important results about the scope and limitations of computability. We give reasons for believing *Church's Thesis*, which essentially says that the theoretical model of computation provided by URM programs is powerful enough to encompass all possible algorithmic computations. From this it follows that our work has applications not just to URM programs but to any conceivable digital computer. This, of course, adds greatly to the significance of the theory. The unit ends with a discussion of some problems that are not algorithmically decidable.

1 URM-COMPUTABLE FUNCTIONS

In this section we extend the notion of URM-computability. First, in Subsection 1.1, we see that there are URM-computable functions which are not primitive recursive. Then, in Subsection 1.2, we introduce the idea of a *partial function* and extend the idea of URM-computability to cover such functions. This extension of URM-computability will prove very important in Section 2.

1.1 A diagonal argument

Recall that the set of primitive recursive functions is the set of functions which we can obtain from the basic primitive recursive functions by the operations of substitution and primitive recursion. We can use the definition of a primitive recursive function to assign to it a *code number*, in a similar way to the method used to code URM programs in *Unit 2*. We use $\delta(f)$ to indicate the code number assigned to the function f in this way.

Strictly speaking, we assign code numbers to *definitions* of primitive recursive functions, rather than to the functions themselves. A given primitive recursive function has lots of definitions (actually, infinitely many) but this will not be significant in our argument.

δ is the Greek letter 'delta'.

First we assign codes to the basic primitive recursive functions. We arrange this so that they all receive codes which are multiples of 3 as follows:

$$\delta(\text{zero}) = 3,$$
$$\delta(\text{succ}) = 9,$$
$$\delta\left(U_m^k\right) = 2^k 3^m \quad \text{for all positive integers } m, k \text{ with } m \leq k.$$

Next, if the function h is defined by substitution from the functions $f, g_1, g_2, \ldots, g_t$ to which we have already assigned code numbers, we put

$$\delta(h) = 2^{\delta(f)} 3^{\delta(g_1)} 5^{\delta(g_2)} \ldots p_{t+1}^{\delta(g_t)} + 1.$$

Remember that p_j is the jth prime number.

Finally, if h is defined from the functions f and g by primitive recursion and we have already assigned codes to f and g, we put

$$\delta(h) = 2^{\delta(f)} 3^{\delta(g)} + 2.$$

In this way we assign a *code number* to every (definition of a) primitive recursive function. Given a natural number m, we can, in principle, decide whether or not m codes a definition of a primitive recursive function and, if so, which definition it codes. In particular, given a natural number m we can decode it and determine whether or not it codes a function $f : \mathbb{N} \longrightarrow \mathbb{N}$, and, if so, we can determine a definition of f showing how it is built up. From this definition, we can compute the values of f. In this way we obtain a function of two variables $\Phi : \mathbb{N}^2 \longrightarrow \mathbb{N}$ such that, for all $m, n \in \mathbb{N}$,

$$\Phi(m,n) = \begin{cases} f(n), & \text{if } m \text{ codes a definition of the primitive} \\ & \text{recursive function } f : \mathbb{N} \longrightarrow \mathbb{N}, \\ 0, & \text{otherwise.} \end{cases}$$

Furthermore, we hope that it is plausible from our description that there is a URM program for computing the values of Φ, that is, Φ is a URM-computable function.

Now we can apply Cantor's diagonal argument to obtain a function which is URM-computable but not primitive recursive. We define $g : \mathbb{N} \longrightarrow \mathbb{N}$ by

See *Unit 2*, Section 2.

$$g(n) = \Phi(n,n) + 1.$$

Given that Φ is URM-computable, it follows that g is URM-computable. Now let f be a primitive recursive function and let m code some definition of f so that, for all $n \in \mathbb{N}$, we have

$$f(n) = \Phi(m,n).$$

Thus $f(m) = \Phi(m,m)$. Since $g(m) = \Phi(m,m) + 1$ we see that $g(m) \neq f(m)$ and hence that $g \neq f$. Thus g is different from each primitive recursive function; that is, g is a URM-computable function which is not primitive recursive.

This argument gives pause for thought. It seems at first sight that if we add a third process for generating URM-computable functions, as suggested in the Introduction, then we could code definitions of functions obtained from the basic functions using substitution, primitive recursion and this third process and repeat the diagonal argument above to obtain a new

URM-computable function different from all those functions generated by the three processes. The way round this difficulty is quite subtle but it has the most profound consequences. Our first step is to refine our ideas about the functions computed by a given URM program P, which we do in the remainder of this section.

1.2 Partial functions and URM-computability

We have seen two ways of looking at URM-computable functions. One way is to take a URM program P and consider the functions it computes. For instance, in *Unit 1* we looked at a URM program P which computes the function add: $\mathbb{N}^2 \longrightarrow \mathbb{N}$. The other way is to look at general ways (such as substitution and primitive recursion) of combining basic URM-computable functions to give more complex URM-computable functions, as we did in *Unit 2* when defining primitive recursive functions. However, in *Unit 1* we deliberately concealed some very important and awkward behaviour which can arise when investigating the functions computed by a given URM program. Now is the time to face up to this behaviour, which we shall illustrate by considering the following URM program.

See Example 2.1 of *Unit 1*.

1 $J(1,2,6)$
2 $S(2)$
3 $S(2)$
4 $S(3)$
5 $J(1,1,1)$
6 $C(3,1)$

It can be seen that a computation using this program carries out the cycle of instructions $1 \to 2 \to 3 \to 4 \to 5 \to 1 \to \cdots$ until, when it carries out the first instruction, the numbers in registers R_1 and R_2 are equal; then the computation jumps to instruction 6 and thereafter halts. In each cycle the number in register R_2 is increased by 2 and that in R_3 is increased by 1. So with input n in register R_1 and 0 in all the other registers, the number in R_2 takes successively the values $0, 2, 4, \ldots$ and the computation halts only when this number is equal to n. Thus we see that a computation with this program with input n halts if and only if n is even, in which case the output is $\frac{1}{2}n$. We are going to extend our definition of URM-computability so that we can assert that this program computes the function f given by

$$f(n) = \begin{cases} \frac{1}{2}n, & \text{if } n \text{ is even,} \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

Up to this point in the course, all our functions of one variable have had domain $\mathbb{N}$, so we shall also have to modify what we mean by a function to cover the case when the domain of a function like this f is a *subset* of $\mathbb{N}$.

We describe a function with domain a subset of $\mathbb{N}$ and codomain $\mathbb{N}$ as a *partial function* from $\mathbb{N}$ to $\mathbb{N}$. If $f : \mathbb{N} \longrightarrow \mathbb{N}$ is a partial function with domain $D \subseteq \mathbb{N}$, we say that $f(n)$ is *undefined* if $n \notin D$. There is an analogous notion of a partial function from $\mathbb{N}^k \longrightarrow \mathbb{N}$. Of course we regard $\mathbb{N}$ as a subset of $\mathbb{N}$, so a function with domain $\mathbb{N}$ and codomain $\mathbb{N}$ will count as a partial function. If we want to emphasize the fact that the domain of f is the whole of $\mathbb{N}$, we shall say that $f : \mathbb{N} \longrightarrow \mathbb{N}$ is a *total function*. We adopt the same convention for functions of more than one variable.

In this context, you can think of '$f(n)$ is defined' as meaning that $f(n)$ has a value in $\mathbb{N}$. 'Undefined' means just that — there is no value of $f(n)$.

Motivated by the example above, we are going to make an important broadening of our notion of URM-computability. Essentially we shall say that a URM program computes a partial function f of one variable if, for input n, the machine halts with output $f(n)$ if $f(n)$ is defined and never halts otherwise. We generalize this to functions of any number of variables as follows.

Definition 1.1 URM-computability

Let P be a URM program and let k be any positive integer. We say that P *computes* the partial function $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ if, for all $(n_1, n_2, \ldots, n_k) \in \mathbb{N}^k$:

(a) if the computation of the URM program P with input $(n_1, n_2, \ldots, n_k)$ halts, it produces the output $f(n_1, n_2, \ldots, n_k)$;

(b) if the computation of the URM program P with input $(n_1, n_2, \ldots, n_k)$ does not halt, $f(n_1, n_2, \ldots, n_k)$ is undefined.

The partial function f is said to be *computed by the URM program* P.

A partial function $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ is said to be *URM-computable* if there is a URM program which computes it.

Henceforth when we say that a function f is URM-computable we shall mean that f is a partial (possibly total) function which is URM-computable in the sense of Definition 1.1.

Recall from *Unit 1* that a given URM program can be used with any finite number of input variables. Thus a given URM program P computes a (partial) function $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ for *each* positive integer k. It will be helpful to use the notation f_P^k for the (partial) function of k variables computed by P.

Example 1.1

Consider the following URM program P.

1 $J(1, 2, 4)$
2 $S(2)$
3 $J(1, 1, 1)$

You can see that this program carries out a cycle of instructions $1 \to 2 \to 3 \to 1 \to \cdots$ until the numbers in registers R_1 and R_2 are equal, in which case the jump to the non-existent instruction 4 means that the computation halts. Of course, if the initial content of R_2 is greater than that of R_1, the $S(2)$ instruction guarantees that the computation never halts, while otherwise the computation does halt with the content of R_1 unchanged.

When we investigate which function of one variable P computes, the input convention for computing such a function makes the initial content of register R_2 equal to 0, so that the computation halts for all inputs n in register R_1. This function, f_P^1, is then given by

$$f_P^1(n) = n, \quad \text{for all } n.$$

In this case the function computed by P is total.

However, if $k \geq 2$, for some inputs $(n_1, n_2, \ldots, n_k)$ we will have $n_1 < n_2$ and the computation will not halt. In general, for $k \geq 2$, the function f_P^k computed by P is given by

$$f_P^k(n_1, n_2, \ldots, n_k) = \begin{cases} n_1, & \text{if } n_1 \geq n_2, \\ \text{undefined}, & \text{otherwise}, \end{cases}$$

and the function is not total. ♦

Problem 1.1

Let P be the URM program with the single instruction

1 $J(1, 2, 1)$

Describe the function f_P^1 of one variable and the function f_P^2 of two variables computed by P. Describe also the functions f_P^k for $k \geq 3$.

Problem 1.2

Show that the partial function $h : \mathbb{N} \longrightarrow \mathbb{N}$ given by

$$h(n) = \begin{cases} \frac{1}{3}n, & \text{if } n \text{ is a multiple of 3,} \\ \text{undefined,} & \text{otherwise,} \end{cases}$$

is URM-computable.

We have now widened our view of what it means for a function to be computed by a URM program. Can we correspondingly widen our understanding of which functions are URM-computable by introducing some extra process besides substitution and primitive recursion? And can we do this in a way for which the diagonal argument in Subsection 1.1 doesn't produce a new sort of URM-computable function not already covered? All this will be answered positively in the next section.

2 RECURSIVE FUNCTIONS

In this section we shall see how a new process for generating (partial) functions, known as *minimization*, can be combined with substitution and primitive recursion to generate all the URM-computable functions, in the new sense defined in Section 1. Subsection 2.1 introduces the concept of minimization, which is closely related to the idea of bounded minimization introduced in *Unit 2*. This leads to the definition of *recursive functions* and to the conclusion that every recursive function is URM-computable. Subsection 2.2 shows how URM-computations can be coded, building on ideas from *Unit 2*, and proves an important theorem concerning this coding. This then, in Subsection 2.3, allows us to prove that all URM-computable functions are recursive, thus showing that URM-computability and recursiveness are equivalent. We also prove some important related results, including *Kleene's Normal Form Theorem.*

2.1 Unbounded searches

As we have seen, a given URM computation may or may not halt and so may or may not produce an output value. Computing a particular value of a function using a URM program involves following the program and carrying out the steps it specifies. We obtain a value for the function if and only if we reach a stage where the computation halts. Recall that this occurs when the next instruction to be carried out does not exist. Thus we can view the computation as a search process. We are searching the stages of the computation to see if there is a stage at which the computation halts because it is directed to carry out a non-existent instruction. Since the computation may not halt, this search is, in principle, unbounded.

We discussed bounded searches in Section 3 of *Unit 2*, where we saw that they produce primitive recursive functions when applied to primitive recursive functions. The observation in the previous paragraph suggests that to capture the behaviour of URM computations we need also to consider *unbounded* searches. When we carry out an unbounded search, there is no guarantee that it will produce a value. Thus it may be difficult to decide algorithmically whether a function defined by an unbounded search is total or not. We shall find that this is the way out of the dilemma caused by the diagonal argument used in Subsection 1.1. We can code definitions of functions defined by unbounded searches but we may not be able to decode numbers to decide whether they correspond to definitions of total functions.

Recall that 'bounded search' is simply an alternative name for 'bounded minimization'.

If so, and we shall see that indeed it is so, this blocks the use of a diagonal argument to try to define a URM-computable function not generated by the processes we are considering.

These considerations lead us to investigate as our third process for generating functions that of unbounded searches. We shall call this process *minimization*. It is an extension of the process of bounded minimization, and we use very similar terminology to define it. We again use the notation μy to stand for 'the minimum y such that'.

Definition 2.1 Minimization on a total function

Let $f : \mathbb{N}^{k+1} \longrightarrow \mathbb{N}$ be a *total* function and let $(n_1, n_2, \ldots, n_k) \in \mathbb{N}^k$ be fixed. We write the *minimization* operation on f as

$$\mu y \, (f(n_1, n_2, \ldots, n_k, y) = 0)$$

and specify its effect as follows:

$$\mu y \, (f(n_1, n_2, \ldots, n_k, y) = 0) = \begin{cases} \text{the smallest } y \in \mathbb{N} \text{ such that} \\ f(n_1, n_2, \ldots, n_k, y) = 0, \\ \text{if there is such a } y, \\ \text{undefined otherwise.} \end{cases}$$

Notice that this definition applies only to *total* functions. It is extended to partial functions later.

Note that if $f : \mathbb{N}^{k+1} \longrightarrow \mathbb{N}$ is a total function and $g : \mathbb{N}^k \longrightarrow \mathbb{N}$ is given by

$$g(n_1, n_2, \ldots, n_k) = \mu y \, (f(n_1, n_2, \ldots, n_k, y) = 0) \tag{2.1}$$

then g will, in general, be a *partial* function.

As in the case of bounded minimization, we can extend the concept to relations.

Definition 2.2 Minimization on a relation

Let $R(n_1, n_2, \ldots, n_k, y)$ be a $(k+1)$-place relation on $\mathbb{N}^{k+1}$ and let $(n_1, n_2, \ldots, n_k) \in \mathbb{N}^k$ be fixed. We write the *minimization* operation on R as

$$\mu y \, R(n_1, n_2, \ldots, n_k, y)$$

and specify its effect as follows:

$$\mu y \, R(n_1, n_2, \ldots, n_k, y) = \begin{cases} \text{the smallest } y \in \mathbb{N} \text{ for which} \\ R(n_1, n_2, \ldots, n_k, y) \text{ holds,} \\ \text{if there is such a } y, \\ \text{undefined otherwise.} \end{cases}$$

Again, if $g : \mathbb{N}^k \longrightarrow \mathbb{N}$ is given by

$$g(n_1, n_2, \ldots, n_k) = \mu y \, R(n_1, n_2, \ldots, n_k, y) \tag{2.2}$$

then g will, in general, be a *partial* function.

Recall, from *Unit 2*, that $R(n_1, n_2, \ldots, n_k, y)$ holds if and only if $\chi_R(n_1, n_2, \ldots, n_k, y) = 1$, where χ_R is the characteristic function of R, which in turn holds if and only if $\overline{\text{sg}}(\chi_R(n_1, n_2, \ldots, n_k, y)) = 0$, where $\overline{\text{sg}}$ is the signum-bar function. Hence we have

$$\mu y \, R(n_1, n_2, \ldots, n_k, y) \iff \mu y \, (\overline{\text{sg}}(\chi_R(n_1, n_2, \ldots, n_k, y)) = 0).$$

Thus, since $\overline{\text{sg}}$ and χ_R are *total* functions, and therefore so is $\overline{\text{sg}} \circ \chi_R$, minimization on a relation is equivalent to minimization on a total function.

Before giving some examples of the use of minimization, we shall introduce an extra piece of notation to help alert us to the possibility that partial functions might not have a value for all inputs. Let $g : \mathbb{N}^k \longrightarrow \mathbb{N}$ and $h : \mathbb{N}^k \longrightarrow \mathbb{N}$ be partial functions. We write

$$g(n_1, n_2, \ldots, n_k) \approx h(n_1, n_2, \ldots, n_k)$$

to mean that either $g(n_1, n_2, \ldots, n_k)$ and $h(n_1, n_2, \ldots, n_k)$ are both defined and equal, or neither is defined. We shall often use this notation when defining a partial function g in terms of a partial function h. For instance, if $h : \mathbb{N} \longrightarrow \mathbb{N}$ is partial function given by

We considered this function in Problem 1.2.

$$h(n) = \begin{cases} \frac{1}{3}n, & \text{if } n \text{ is a multiple of 3,} \\ \text{undefined,} & \text{otherwise,} \end{cases}$$

we can define a partial function $g : \mathbb{N} \longrightarrow \mathbb{N}$ by

$$g(n) \approx h(n+1), \quad \text{for all } n,$$

so that

$$g(n) = \begin{cases} \frac{1}{3}(n+1), & \text{if } n+1 \text{ is a multiple of 3,} \\ \text{undefined,} & \text{otherwise.} \end{cases}$$

Also, now that we have this new notation, we could rewrite (2.1) and (2.2) as

$$g(n_1, n_2, \ldots, n_k) \approx \mu y \, (f(n_1, n_2, \ldots, n_k, y) = 0),$$

$$g(n_1, n_2, \ldots, n_k) \approx \mu y \, R(n_1, n_2, \ldots, n_k, y).$$

Using this notation we can say that two partial functions $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ and $g : \mathbb{N}^k \longrightarrow \mathbb{N}$ are *equal*, and write $f = g$, if, for all $(n_1, n_2, \ldots, n_k) \in \mathbb{N}^k$,

$$f(n_1, n_2, \ldots, n_k) \approx g(n_1, n_2, \ldots, n_k).$$

Example 2.1

Let $g : \mathbb{N} \longrightarrow \mathbb{N}$ be the function given by

$$g(n) \approx \mu y \, (y^2 = n).$$

Thus we have $g(0) = 0$, $g(1) = 1$, $g(2)$ and $g(3)$ are not defined, $g(4) = 2$ and so on. Thus, for each $n \in \mathbb{N}$,

$$g(n) = \begin{cases} \sqrt{n}, & \text{if } n \text{ is a square,} \\ \text{undefined,} & \text{otherwise.} \end{cases}$$

♦

Example 2.2

Let $g : \mathbb{N} \longrightarrow \mathbb{N}$ be the function given by

$$g(n) \approx \mu y \, (n \dot{-} 3y = 0).$$

We observe that $n \dot{-} 3y = 0$ if and only if $n \leq 3y$ and hence, for each n, $g(n) \approx \mu y \, (n \leq 3y)$; that is, for each n, $g(n)$ is the smallest y such that $n \leq 3y$. As there is such a y for all n, it follows that g is a total function, with $g(0) = 0$, $g(1) = 1$, $g(2) = 1$, $g(3) = 1$, $g(4) = 2$ and so on. ♦

In this case, g could actually be defined by bounded minimization, as the number n itself provides a suitable bound on y. We have

$$g(n) = \mu y \leq n \, (n \dot{-} 3y = 0).$$

In the following and subsequent problems, when we ask you to 'evaluate $g(n)$', where g is a partial function, we mean 'determine whether $g(n)$ is defined and, if so, state its value'.

Problem 2.1

Let $g : \mathbb{N} \longrightarrow \mathbb{N}$ be the function given by

$$g(n) \approx \mu y \, (2y = n).$$

Evaluate $g(n)$ for $0 \leq n \leq 6$.

Problem 2.2

Let $g : \mathbb{N} \longrightarrow \mathbb{N}$ be the function given by

$$g(n) \approx \mu y\,(n \dot{-} y^2 = 0).$$

Evaluate $g(n)$ for $0 \leq n \leq 9$.

In Definition 2.1, we specified that the function f should be a total function. Now we consider how to apply minimization to partial functions. We need to take some care with this definition if we wish minimization to lead from URM-computable functions to URM-computable functions. Suppose, for example, that $f : \mathbb{N}^2 \longrightarrow \mathbb{N}$ is a URM-computable function and that we have

$f(3,0) = 1,$
$f(3,1) = 5,$
$f(3,2)$ is undefined,
$f(3,3) = 7,$
$f(3,4) = 0.$

Now consider how we would set out to compute $\mu y\,(f(3,y) = 0)$. The natural way to do this would be to use the URM program which computes f to obtain successively the values of $f(3,0)$, $f(3,1)$, $f(3,2)$, $f(3,3)$, $f(3,4)$ and so on, until we found a value of y for which $f(3,y) = 0$. However, in this case, as $f(3,2)$ is undefined, when we come to the input $(3,2)$ the computation will not halt. So we would never reach the stage of inputting $(3,3)$ and then $(3,4)$. Thus we would never discover that $f(3,4) = 0$. Hence, contrary to what might be expected from the table of values above, our computation would not halt with output 4. We should therefore be bound to say that $\mu y\,(f(3,y) = 0)$ is undefined. We could detect 4 as the smallest value of y for which $f(3,y) = 0$ only if the computations of the value $f(3,y)$ for $0 \leq y < 4$ all halted and produced non-zero outputs in each case.

This example suggests (correctly, as we shall see later) that if we want minimization applied to URM-computable functions to yield URM-computable functions, then we need to define minimization applied to partial functions in the following way.

Definition 2.3 Minimization on a partial function

Let $f : \mathbb{N}^{k+1} \longrightarrow \mathbb{N}$ be a *partial* function and let $(n_1, n_2, \ldots, n_k) \in \mathbb{N}^k$ be fixed. We write the *minimization* operation on f as

$$\mu y\,(f(n_1, n_2, \ldots, n_k, y) = 0)$$

and specify its effect as follows:

$$\mu y\,(f(n_1, n_2, \ldots, n_k, y) = 0) = \begin{cases} z, & \text{if } f(n_1, n_2, \ldots, n_k, z) = 0 \\ & \text{and} f(n_1, n_2, \ldots, n_k, y) \\ & \text{is defined and} \\ & f(n_1, n_2, \ldots, n_k, y) \neq 0 \\ & \text{for all } y \text{ such that} \\ & 0 \leq y < z, \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

The partial function

$$g(n_1, n_2, \ldots, n_k) \approx \mu y\,(f(n_1, n_2, \ldots, n_k, y) = 0)$$

defined in this way is said to be obtained from f by *minimization*.

In the case where the function f is total, this definition accords with Definition 2.1.

Example 2.3

Let $f : \mathbb{N}^2 \longrightarrow \mathbb{N}$ be the partial function given by

$$f(n,y) = \begin{cases} 0, & \text{if } n \neq 0, y \neq 0 \text{ and } n \text{ is a divisor of } y, \\ 1, & \text{if } n = 0 \text{ or } y = 0, \\ \text{undefined}, & \text{otherwise,} \end{cases}$$

and let $g : \mathbb{N} \longrightarrow \mathbb{N}$ be obtained from f by minimization. Thus

$$g(n) \approx \mu y\,(f(n,y) = 0).$$

We have the following table of values of f, for $0 \leq n \leq 6$ and $0 \leq y \leq 6$, where $*$ means 'undefined'.

		y						
$f(n,y)$		0	1	2	3	4	5	6
	0	1	1	1	1	1	1	1
	1	1	0	0	0	0	0	0
	2	1	$*$	0	$*$	0	$*$	0
n	3	1	$*$	$*$	0	$*$	$*$	0
	4	1	$*$	$*$	$*$	0	$*$	$*$
	5	1	$*$	$*$	$*$	$*$	0	$*$
	6	1	$*$	$*$	$*$	$*$	$*$	0

We see that:

(a) $g(0)$ is not defined because there is no $y \in \mathbb{N}$ with $f(0,y) = 0$;

(b) $g(1) = 1$, as $f(1,0)$ is defined and is non-zero and $f(1,1) = 0$;

(c) for $n > 1$, $g(n)$ is not defined as in this case $y = n$ gives the smallest value of y for which $f(n,y) = 0$, but $f(n,n-1)$ is not defined.

To summarize,

$$g(n) = \begin{cases} 1, & \text{if } n = 1, \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

♦

Problem 2.3

Let $f : \mathbb{N}^2 \longrightarrow \mathbb{N}$ be the partial function given by

$$f(n,y) = \begin{cases} n - y^2, & \text{if } n \geq y^2, \\ \text{undefined}, & \text{otherwise,} \end{cases}$$

and let $g : \mathbb{N} \longrightarrow \mathbb{N}$ be the partial function given by

$$g(n) \approx \mu y\,(f(n,y) = 0).$$

(a) Evaluate $g(2)$ and $g(4)$.

(b) What is the domain of the function g?

Problem 2.4

Values of the partial function $f : \mathbb{N}^2 \longrightarrow \mathbb{N}$ are shown in the following table, where $*$ means 'undefined'.

		y						
$f(n,y)$		0	1	2	3	4	5	6
	0	3	2	$*$	0	1	$*$	0
	1	4	3	0	$*$	$*$	0	1
	2	4	2	7	$*$	0	2	3
n	3	0	$*$	$*$	$*$	$*$	$*$	$*$
	4	2	3	7	0	$*$	$*$	$*$
	5	2	$*$	2	3	4	5	6
	6	2	0	$*$	$*$	1	2	3

Let $g : \mathbb{N} \longrightarrow \mathbb{N}$ be the partial function given by

$$g(n) \approx \mu y\,(f(n, y) = 0).$$

Evaluate $g(n)$ for $0 \leq n \leq 6$.

It is worth noting that the difficulty we encountered when extending the minimization operation from total to partial functions does not apply in the case of relations since, as we saw earlier, minimization on a relation is equivalent to minimization on a total function.

We are going to study the class of functions that we get when we add minimization to the operations which we used to generate the primitive recursive functions. Since minimization will, in general, produce a partial function even when applied to a total function, we need to extend the definitions of substitution and primitive recursion to partial functions.

Substitution If $f : \mathbb{N}^t \longrightarrow \mathbb{N}$, $g_1 : \mathbb{N}^k \longrightarrow \mathbb{N}$, $g_2 : \mathbb{N}^k \longrightarrow \mathbb{N}, \ldots, g_t : \mathbb{N}^k \longrightarrow \mathbb{N}$ are partial functions, then the partial function $h : \mathbb{N}^k \longrightarrow \mathbb{N}$ such that

$$h(n_1, n_2, \ldots, n_k) \approx$$

$$f(g_1(n_1, n_2, \ldots, n_k), g_2(n_1, n_2, \ldots, n_k), \ldots, g_t(n_1, n_2, \ldots, n_k)),$$

is said to be obtained from $f, g_1, g_2, \ldots, g_t$ by *substitution.*

$h(n_1, \ldots, n_k)$ is defined only when $g_1(n_1, \ldots, n_k), \ldots, g_t(n_1, \ldots, n_k)$ are all defined and $f(g_1(n_1, \ldots, n_k), \ldots, g_t(n_1, \ldots, n_k))$ is defined.

Primitive recursion If $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ and $g : \mathbb{N}^{k+2} \longrightarrow \mathbb{N}$ are partial functions, then the partial function $h : \mathbb{N}^{k+1} \longrightarrow \mathbb{N}$ such that

$$h(n_1, n_2, \ldots, n_k, 0) \approx f(n_1, n_2, \ldots, n_k),$$

$$h(n_1, n_2, \ldots, n_k, n+1) \approx g(n_1, n_2, \ldots, n_k, n, h(n_1, n_2, \ldots, n_k, n)),$$

is said to be obtained from f and g by *primitive recursion.*

$h(n_1, \ldots, n_k, n+1)$ is defined only when $h(n_1, \ldots, n_k, n)$ is defined and $g(n_1, \ldots, n_k, n, h(n_1, \ldots, n_k, n))$ is defined.

Example 2.4

Let $g_1 : \mathbb{N} \longrightarrow \mathbb{N}$ and $g_2 : \mathbb{N} \longrightarrow \mathbb{N}$ be the partial functions given by

$$g_1(n) = \begin{cases} n, & \text{if } n \text{ is a multiple of 2,} \\ \text{undefined,} & \text{otherwise,} \end{cases}$$

$$g_2(n) = \begin{cases} n+8, & \text{if } n \text{ is a multiple of 3,} \\ \text{undefined,} & \text{otherwise,} \end{cases}$$

and let $f : \mathbb{N}^2 \longrightarrow \mathbb{N}$ be the partial function given by

$$f(n_1, n_2) = \begin{cases} n_1 + 1, & \text{if both } n_1 \text{ and } n_2 \text{ are multiples of 4,} \\ \text{undefined.} & \text{otherwise,} \end{cases}$$

We shall describe the function $h : \mathbb{N} \longrightarrow \mathbb{N}$ obtained from f, g_1, g_2 by substitution.

For $h(n)$ to be defined, we need both $g_1(n)$ and $g_2(n)$ to be defined, which only happens when n is divisible by both 2 and 3, i.e. when n is divisible by 6, in which case $g_1(n) = n$ and $g_2(n) = n + 8$. In addition we need $f(g_1(n), g_2(n))$ to be defined. Putting $n = 6k$, where k is a natural number, this requires both $g_1(n) = 6k$ and $g_2(n) = 6k + 8$ to be divisible by 4, which only happens when k is even, so that n is divisible by 12. Thus h is the function

$$h(n) = \begin{cases} n+1, & \text{if } n \text{ is a multiple of 12,} \\ \text{undefined,} & \text{otherwise.} \end{cases}$$

♦

Problem 2.5

Let $f\colon \mathbb{N} \longrightarrow \mathbb{N}$ and $g\colon \mathbb{N}^3 \longrightarrow \mathbb{N}$ be the functions defined by

$$f(n) = 1, \quad \text{for all } n,$$

$$g(n_1, n, m) = \begin{cases} n_1 + m, & \text{if } m \leq 2, \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

Describe the function $h\colon \mathbb{N}^2 \longrightarrow \mathbb{N}$ obtained from f and g by primitive recursion.

The class of primitive recursive functions was introduced in Definition 1.2 of *Unit 2*. We now extend that class to our most important class of functions.

Definition 2.4 Recursive function

A function is said to be *recursive* if it can be obtained from basic primitive recursive functions using the operations of substitution, primitive recursion and minimization on a function a finite number of times.

The basic primitive recursive functions are given in Definition 1.1 of *Unit 2*.

We shall often find it convenient to deal with relations as well as functions.

Definition 2.5 Recursive relation

A relation R is said to be a *recursive relation* if its characteristic function χ_R is a recursive function.

We saw earlier that minimization on a relation R is equivalent to minimization on the total function $\overline{\text{sg}} \circ \chi_R$. Furthermore, if the relation R is recursive then so is the function χ_R and hence, since $\overline{\text{sg}}$ is primitive recursive, so is the function $\overline{\text{sg}} \circ \chi_R$. Therefore, it is an immediate consequence of Definitions 2.4 and 2.5 that if a function f can be obtained from functions and relations already known to be recursive by substitution, primitive recursion or minimization (on a function or relation) then f is recursive.

Problem 2.6

Show that the partial function $h : \mathbb{N} \longrightarrow \mathbb{N}$ given by

$$h(n) = \begin{cases} \frac{1}{3}n, & \text{if } n \text{ is a multiple of 3,} \\ \text{undefined}, & \text{otherwise,} \end{cases}$$

is recursive.

We shall also have need for recursive sets.

Definition 2.6 Recursive set

A subset A of $\mathbb{N}$ is said to be a *recursive set* if its characteristic function χ_A is a recursive function.

In general, a recursive function is a partial function. A recursive function which is actually a total function is called a *total recursive function.*

Theorem 2.1

Every primitive recursive function is a total recursive function.

Proof

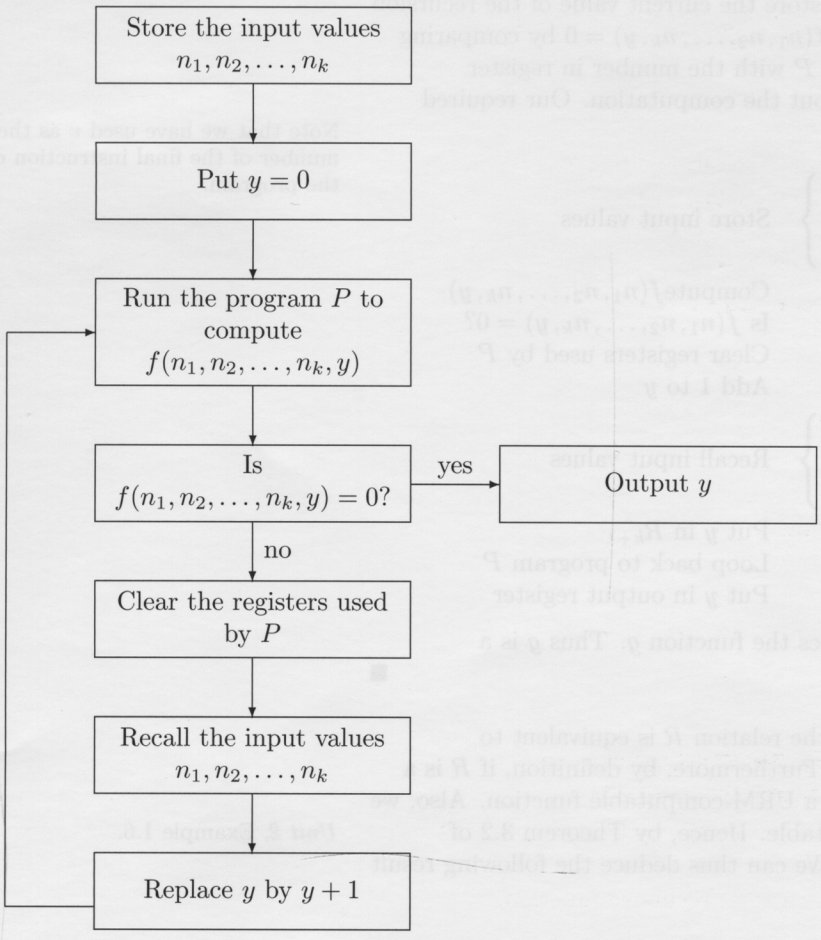
A primitive recursive function is a total function. Since the processes for generating recursive functions include those for generating primitive recursive functions, a primitive recursive function is recursive. ■

Now we turn our attention to the relationship between the recursive functions and the URM-computable functions. We shall show that these two classes of functions are indeed identical. First we show that every recursive function is URM-computable.

We know that the basic primitive recursive functions are URM-computable. It is relatively easily to extend the proofs of Theorems 3.1–3.4 of *Unit 1* to show that our wider class of (partial) URM-computable functions is closed under the operations of substitution and primitive recursion. Thus all that remains to be done is to show that the class of URM-computable functions is also closed under the operation of minimization.

Since the proof of the closure of this wider class under substitution and primitive recursion follows very similar lines to the proofs for the narrower class in *Unit 1*, Theorems 3.1–3.4, we omit the details.

It should be reasonably clear from our earlier discussion that if we have a URM program for computing the values of a partial function $f : \mathbb{N}^{k+1} \longrightarrow \mathbb{N}$, then we can derive from it a URM program for computing $\mu y\,(f(n_1, n_2, \ldots, n_k, y) = 0)$. The required program will compute successively the values $f(n_1, n_2, \ldots, n_k, 0)$, $f(n_1, n_2, \ldots, n_k, 1)$, $f(n_1, n_2, \ldots, n_k, 2), \ldots$ until it obtains the value 0 or until it reaches an n for which $f(n_1, n_2, \ldots, n_k, n)$ is undefined (in which case the program makes no further progress and fails to halt). There are some technical details to take care of such as storing the input values $n_1, n_2, \ldots, n_k$ in registers not used in the computation of f, keeping track of the current value of the recursion variable y and clearing all appropriate registers before computing the next value of f. We shall deal with these technicalities later. First we give a flow chart to indicate how our program is intended to operate. We assume that f is a URM-computable (so, in general, a partial) function which is computed by the URM program P.



If we reach a value of y for which $f(n_1, n_2, \ldots, n_k, y)$ is not defined, the computation using the program P for this value of y does not terminate and we do not obtain an output value. Likewise if f is a total function but there is no $y \in \mathbb{N}$ such that $f(n_1, n_2, \ldots, n_k, y) = 0$, then the program goes round the loop indefinitely and again there is no output. This is exactly what is specified for $\mu y\,(f(n_1, n_2, \ldots, n_k, y) = 0)$ in such cases. We shall show how to convert the flow chart into a program.

Recall from *Unit 1* that we use $\rho(P)$ for the largest number u such that the URM program P uses the register R_u. Recall also from *Unit 1* that we use $Z(1, m)$ to represent the program

$$\begin{array}{ll} 1 & Z(1) \\ 2 & Z(2) \\ \vdots & \vdots \\ m & Z(m) \end{array}$$

which clears all the registers $R_1, R_2, \ldots, R_m$.

Theorem 2.2 Closure under minimization on a function

If $f : \mathbb{N}^{k+1} \longrightarrow \mathbb{N}$ is a URM-computable function, then so also is the function $g : \mathbb{N}^k \longrightarrow \mathbb{N}$ given by

$$g(n_1, n_2, \ldots, n_k) \approx \mu y\,(f(n_1, n_2, \ldots, n_k, y) = 0).$$

Proof

Let $f : \mathbb{N}^{k+1} \longrightarrow \mathbb{N}$ be a URM-computable function and let P be a URM program which computes the function f. Suppose that $\rho(P) = u$. We can use the registers $R_{u+1}, R_{u+2}, \ldots, R_{u+k}$ to store the input values $n_1, n_2, \ldots, n_k$ and register R_{u+k+1} to store the current value of the recursion variable y. We check whether or not $f(n_1, n_2, \ldots, n_k, y) = 0$ by comparing the output from running the program P with the number in register R_{u+k+2}, which will remain 0 throughout the computation. Our required program is as follows.

Note that we have used v as the number of the final instruction of the program.

1	$C(1, u+1)$	} Store input values
$\vdots$	$\vdots$	
k	$C(k, u+k)$	
	P	Compute $f(n_1, n_2, \ldots, n_k, y)$
	$J(1, u+k+2, v)$	Is $f(n_1, n_2, \ldots, n_k, y) = 0$?
	$Z(1, u)$	Clear registers used by P
	$S(u+k+1)$	Add 1 to y
	$C(u+1, 1)$	} Recall input values
	$\vdots$	
	$C(u+k, k)$	
	$C(u+k+1, k+1)$	Put y in R_{k+1}
	$J(1, 1, k+1)$	Loop back to program P
v	$C(u+k+1, 1)$	Put y in output register

We can see that this program computes the function g. Thus g is a URM-computable function. ■

We saw earlier that minimization on the relation R is equivalent to minimization on the function $\overline{\text{sg}} \circ \chi_R$. Furthermore, by definition, if R is a URM-computable relation then χ_R is a URM-computable function. Also, we saw in *Unit 2* that $\overline{\text{sg}}$ is URM-computable. Hence, by Theorem 3.2 of *Unit 1*, $\overline{\text{sg}} \circ \chi_R$ is URM-computable. We can thus deduce the following result from Theorem 2.2.

Unit 2, Example 1.6.

Theorem 2.3 Closure under minimization on a relation

If R is a URM-computable $(k+1)$-place relation, then the function $g : \mathbb{N}^k \longrightarrow \mathbb{N}$ given by

$$g(n_1, n_2, \ldots, n_k) \approx \mu y \; R(n_1, n_2, \ldots, n_k, y)$$

is a URM-computable function.

From Theorems 2.2 and 2.3 and the corresponding closure results for substitution and primitive recursion, it follows that all the functions obtainable from the basic primitive recursive functions using the operations of substitution, primitive recursion and minimization (on a function or relation) are URM-computable. In other words, we have the following important theorem.

Theorem 2.4

Every recursive function is URM-computable.

Example 2.5

Let P be the following program.

1 $J(2,3,6)$
2 $S(4)$
3 $S(4)$
4 $S(3)$
5 $J(1,1,1)$
6 $S(4)$
7 $J(1,4,11)$
8 $Z(1)$
9 $S(1)$
10 $J(1,1,12)$
11 $Z(1)$

You may like to check that this program computes the function $f : \mathbb{N}^2 \longrightarrow \mathbb{N}$ given by

$$f(n,y) = \begin{cases} 0, & \text{if } n = 2y+1, \\ 1, & \text{otherwise.} \end{cases}$$

Now let $g : \mathbb{N} \longrightarrow \mathbb{N}$ be the partial function given by

$$g(n) \approx \mu y \, (f(n,y) = 0).$$

Thus

$$g(n) = \begin{cases} \frac{1}{2}(n-1), & \text{if } n \text{ is odd}, \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

It would not be difficult to devise a URM program which computes g directly. However, to illustrate the method used in our proof of Theorem 2.2, we give the program that we would obtain from P following this method.

In this case $u = 4$ as P uses the registers R_1, R_2, R_3 and R_4 only. Also, $k = 1$ here. Thus the program we obtain is as follows.

1 $C(1,5)$
2 $J(2,3,7)$
3 $S(4)$
4 $S(4)$
5 $S(3)$
6 $J(1,1,2)$
7 $S(4)$
8 $J(1,4,12)$
9 $Z(1)$
10 $S(1)$
11 $J(1,1,13)$
12 $Z(1)$
13 $J(1,7,22)$
14 $Z(1)$
15 $Z(2)$
16 $Z(3)$
17 $Z(4)$
18 $S(6)$
19 $C(5,1)$
20 $C(6,2)$
21 $J(1,1,2)$
22 $C(6,1)$

♦

Problem 2.7

Let $f : \mathbb{N}^2 \longrightarrow \mathbb{N}$ be the function computed by the following program.

1 $J(2,3,15)$
2 $S(3)$
3 $J(2,3,15)$
4 $Z(3)$
5 $J(1,3,12)$
6 $J(2,4,10)$
7 $S(3)$
8 $S(4)$
9 $J(1,1,5)$
10 $Z(4)$
11 $J(1,1,5)$
12 $J(2,4,14)$
13 $J(1,1,15)$
14 $Z(1)$

Give the program, as derived from the proof of Theorem 2.2, which computes $\mu y\,(f(n,y) = 0)$.

Problem 2.8

Suppose that the URM program P computes a function $f : \mathbb{N}^{k+1} \longrightarrow \mathbb{N}$ of $k+1$ variables and that P has r instructions and uses the registers $R_1, R_2, \ldots, R_u$. Let P^* be the URM program, derived from the proof of Theorem 2.2, which computes the function $\mu y\,(f(n_1, n_2, \ldots, n_k, y) = 0)$. How many instructions does the program P^* have?

We have seen that recursive functions are URM-computable. Also we have given some reasons to suggest that adding minimization to the other operations allows us to generate *all* the URM-computable functions. Indeed this is the case, and we turn our attention to its proof in the remainder of this section.

2.2 Coding URM computations

Having shown, in Theorem 2.4, that every recursive function is URM-computable, we now turn to proving that every URM-computable function is recursive. The idea of the proof is quite straightforward. Carrying out the proof in detail involves a number of technicalities but you should not let these obscure the main idea.

The main idea is this. We show that the stages of a URM computation can be coded using primitive recursive functions. We have already made a start on this in Section 3 of *Unit 2*, where we showed that various functions used in connection with the coding of URM programs are primitive recursive. Obtaining an output value for the function involves running through the stages of the computation until it halts and then, if we find such a stage, extracting the output value as the number in register R_1 when the computation halts.

Searching for a stage when the computation halts corresponds to minimization and it will be easy to see that, when such a stage is reached, the output value can be extracted using another primitive recursive function. Thus the values of the function can be obtained from primitive recursive functions by means of minimization. Then it follows at once that the function computed by the program is recursive.

To check in detail that a proof on these lines can be carried out, we need to specify a particular way of coding URM computations. We begin by considering what we mean by the *stages* of a URM computation. These correspond to the rows of the trace table of the computation. Here it is useful to look again at Example 1.1 of *Unit 1*.

We began with the following program.

1 $J(2,3,5)$
2 $S(1)$
3 $S(3)$
4 $J(1,1,1)$

We found that the trace table for the computation using this program, with input $(7,2)$, is as follows.

Stage	Instruction	R_1	R_2	R_3
0	1	7	2	0
1	2	7	2	0
2	3	8	2	0
3	4	8	2	1
4	1	8	2	1
5	2	8	2	1
6	3	9	2	1
7	4	9	2	2
8	1	9	2	2
9	5	9	2	2

You will notice that we have modified the trace table as given in the original example in two ways. First we have added an extra column headed 'Stage' which we use to count the rows in the table. Thus each stage corresponds to carrying out one instruction. Second we have labelled the instruction at stage 9 with a 5 rather than labelling it with 'STOP' as we did earlier. Of course these labels amount to the same thing: the program has only 4 instructions so, when at stage 8 it carries out instruction 1 and jumps to the non-existent instruction 5, this is the same as the computation stopping.

Note that the initial stage is labelled 0, so stage t is reached after t instructions have been carried out. The state, or *situation*, of the computation at each stage is fully described by a sequence of numbers, the

first of which is the number of the instruction about to be carried out and the remainder are the numbers stored at that stage in the registers used by the program. This row of numbers can be coded using prime powers in the same way as we coded sequences in *Unit 2*.

Suppose that, at a given stage, the number of the instruction to be carried out is a and the numbers in the registers used by the program are $r_1, r_2, \ldots, r_b$. We then say that the situation at that stage is coded by the number

$$s = p_1^a p_2^{r_1} p_3^{r_2} \ldots p_{b+1}^{r_b}$$

where, as before, p_j is the jth prime number. We call s the *situation number* at that stage.

Example 2.6

We add, to the trace table given above, the situation number at each stage of the computation.

Stage	Instruction	R_1	R_2	R_3	Situation number
0	1	7	2	0	$2^1 3^7 5^2 7^0 = 109\,350$
1	2	7	2	0	$2^2 3^7 5^2 7^0 = 218\,700$
2	3	8	2	0	$2^3 3^8 5^2 7^0 = 1312\,200$
3	4	8	2	1	$2^4 3^8 5^2 7^1 = 18\,370\,800$
4	1	8	2	1	$2^1 3^8 5^2 7^1 = 2296\,350$
5	2	8	2	1	$2^2 3^8 5^2 7^1 = 4592\,700$
6	3	9	2	1	$2^3 3^9 5^2 7^1 = 27\,556\,200$
7	4	9	2	2	$2^4 3^9 5^2 7^2 = 385\,786\,800$
8	1	9	2	2	$2^1 3^9 5^2 7^2 = 48\,223\,350$
9	5	9	2	2	$2^5 3^9 5^2 7^2 = 771\,573\,600$

♦

Problem 2.9

Let P be the following URM program.

1 $J(1,2,5)$
2 $S(2)$
3 $S(3)$
4 $J(1,1,1)$
5 $C(3,1)$

Write down the trace table for the computation using the program P with input $(4,2)$ and calculate the situation number at each stage of the computation.

If we know the situation number at a given stage of a computation then we can determine the number of the instruction to be performed and the contents of the registers at this stage. Recall from *Unit 2* that if n, j are positive integers and $n \geq 2$ then $(n)_j$ is the exponent of the jth prime p_j in the prime decomposition of n, if p_j appears in that decomposition, and is 0 otherwise. Now suppose that the situation number at some stage of a computation is s. Then the number a of the instruction to be performed at this stage is given by $a = (s)_1$: that is, the exponent of $p_1 = 2$ in the prime factorization of s. Also the content of register R_c is $(s)_{c+1}$. In particular, since the function $(n, j) \longmapsto (n)_j$ is primitive recursive, the number a of the instruction to be performed and the contents of the registers may be calculated from s using a primitive recursive function.

See Theorem 3.8 of *Unit 2*.

The situation number s will depend on the particular URM program that we are using, the numbers we are using as input and the stage in the computation we have reached. We have already seen in Section 2 of *Unit 2* how to code URM programs by numbers. Recall that we use $\gamma(P)$ for the code number of the URM program P. Not every number codes a URM

program, but we saw in Theorem 3.10 of *Unit 2* that the set *Prog* of code numbers of URM programs is primitive recursive. We should also recall that a given URM program will compute different functions according to how many variables we allow as inputs. We are led to consider, for each positive integer k, the function

$$S_k : \mathbb{N}^{k+2} \longrightarrow \mathbb{N}$$

defined as follows:

$$S_k(e, n_1, n_2, \ldots n_k, t) = \begin{cases} \text{the situation number at stage } t \\ \text{of the URM computation using} \\ \text{the URM program } P \text{ with input} \\ (n_1, n_2, \ldots, n_k), \text{ if } \gamma(P) = e, \\ 0, \text{ if } e \text{ does not code a URM program.} \end{cases}$$

Our aim is to show that this function S_k is primitive recursive, but we have to deal with one small technical problem. If S_k is to be a primitive recursive function then it must be a total function: that is, it must be defined for all values of $e, n_1, n_2, \ldots, n_k, t \in \mathbb{N}$. If, however, a given computation halts, then t can take only a finite number of values. For example, in the trace table above we have stages 0 to 9 only. Of course if the computation does not terminate then $S_k(e, n_1, n_2, \ldots, n_k, t)$ is defined for all $t \in \mathbb{N}$. We get over this difficulty in a natural way. If a computation terminates, then we suppose that it remains in its final situation thereafter. Thus, if the computation terminates at stage t_0, then we put
$S_k(e, n_1, n_2, \ldots, n_k, t) = S_k(e, n_1, n_2, \ldots, n_k, t_0)$ for all $t > t_0$.

Example 2.7

Let e be the number which codes the URM program of Example 2.6. Then from the situation numbers which we calculated in that example we see that, for example,

$$S_2(e, 7, 2, 0) = 109\,350,$$
$$S_2(e, 7, 2, 4) = 2296\,350,$$
$$S_2(e, 7, 2, t) = 771\,573\,600, \quad \text{for all } t \geq 9.$$ ♦

Problem 2.10

Let e be the number which codes the URM program of Problem 2.9. Give the values of the situation numbers $S_2(e, 4, 2, 0)$, $S_2(e, 4, 2, 9)$ and $S_2(e, 4, 2, 100)$.

Problem 2.11

Let

$$e = 2^{281252}3^{12}5^{12}7^{18}11^{32}13^{25}.$$

Calculate the values of $S_1(e, 4, t)$ for $0 \leq t \leq 20$.

In your calculations you will have noticed that, from the situation number at a given stage, the situation number at the next stage is easily obtained. We exploit this in our proof of Theorem 2.5, which shows that S_k is a primitive recursive function and which has profound consequences.

Preamble to Theorem 2.5

Suppose that e codes a URM program P and $S_k(e, n_1, n_2, \ldots, n_k, t)$ is the situation number at stage t of a computation using the program P with input $(n_1, n_2, \ldots, n_k)$ and suppose that the computation has not halted at this stage. From e and $S_k(e, n_1, n_2, \ldots, n_k, t)$ we can extract the information needed to compute $S_k(e, n_1, n_2, \ldots, n_k, t + 1)$.

The number a of the instruction to be performed at stage t is given by $a = (S_k(e, n_1, n_2, \ldots, n_k, t))_1$: that is, the highest power of 2 that divides $S_k(e, n_1, n_2, \ldots, n_k, t)$. Moreover the code number c of the instruction a is the highest power of p_a which divides e: that is, $c = (e)_a$. Thus, from the code number c, we can determine the nature of the instruction to be performed at stage t. Since we can extract from $S_k(e, n_1, n_2, \ldots, n_k, t)$ the contents at stage t of the registers used by P, we can then determine the situation number $S_k(e, n_1, n_2, \ldots, n_k, t+1)$ at stage $t+1$. These considerations suggest that S_k can be defined by primitive recursion.

Theorem 2.5

For each $k \geq 1$, the function S_k is primitive recursive.

Proof

We shall show that the function is primitive recursive by showing, as suggested in the preamble to the theorem, that it can be defined by primitive recursion.

The first equation in the primitive recursive definition is easily obtained. At stage 0 we are about to carry out instruction number 1 with the input numbers $n_1, n_2, \ldots, n_k$ in the first k registers. So the situation number does not depend on the particular program and we have

$$S_k(e, n_1, n_2, \ldots, n_k, 0) = \begin{cases} 2^1 3^{n_1} 5^{n_2} \ldots p_{k+1}^{n_k}, & \text{if } e \in \mathit{Prog}, \\ 0, & \text{otherwise,} \end{cases}$$

where *Prog* is the set of code numbers of URM programs. *Prog* is a primitive recursive set (*Unit 2*, Theorem 3.10), hence using results from *Unit 2* we can deduce that the relations defining the cases are primitive recursive. Furthermore, using other results from *Unit 2* we can deduce that the functions given by the formulas $2^1 3^{n_1} 5^{n_2} \ldots p_{k+1}^{n_k}$ and 0 on the right-hand side are primitive recursive. Therefore, by Theorem 1.5 of *Unit 2*, the function defined by cases giving the values of $S_k(e, n_1, n_2, \ldots, n_k, 0)$ is primitive recursive.

In particular, we use the facts that the functions mult and exp are primitive recursive, and that the function p which enumerates the prime numbers is primitive recursive (*Unit 2*, Theorem 3.6).

Now we have to think about the recursion equation. In other words we have to express $S_k(e, n_1, n_2, \ldots, n_k, t+1)$ in terms of $e, n_1, n_2, \ldots, n_k, t$ and $S_k(e, n_1, n_2, \ldots, n_k, t)$ using known primitive recursive functions. The preamble indicates that we can calculate $S_k(e, n_1, n_2, \ldots, n_k, t+1)$ from e and $S_k(e, n_1, n_2, \ldots, n_k, t)$ in the case $e \in \mathit{Prog}$. It is not difficult to use this idea to provide the recursion equation but it is technically complicated, as we need to consider a number of cases, and so we shall not give all the details.

The recursion equation will be defined by cases (Section 1 of *Unit 2*). We have six different cases to consider.

(i) e does not code a URM program.

(ii) e codes a URM program and the computation halts at stage t or earlier.

(iii) e codes a URM program and the instruction to be carried out at stage t is a Zero instruction.

(iv) e codes a URM program and the instruction to be carried out at stage t is a Successor instruction.

(v) e codes a URM program and the instruction to be carried out at stage t is a Copy instruction.

(vi) e codes a URM program and the instruction to be carried out at stage t is a Jump instruction.

These cases are clearly mutually exclusive and exhaustive.

The first step is to check that each of these cases corresponds to a primitive recursive relation.

(i) The set *Prog* of code numbers of URM programs is primitive recursive, so its complement is also primitive recursive (*Unit 2*, Theorem 1.2). Hence (i) is a primitive recursive relation.

(ii) From the definition of situation numbers we see that, if a computation halts at stage t or earlier, then the number of the instruction to be carried out at stage t is greater than the number of instructions in the program. The number of instructions in the program with code number e is $\text{len}(e)$, where len is the primitive recursive function which gives the length of a sequence coded by a number. Also, from the preamble, the instruction to be carried out at stage t is $(S_k(e, n_1, n_2, \ldots, n_k, t))_1$, which is obtained from $S_k(e, n_1, n_2, \ldots, n_k, t)$ using the primitive recursive function $(n, j) \longmapsto (n)_j$. Hence relation (ii) can be expressed as

See *Unit 2*, Subsection 3.2.

$$e \in \mathit{Prog} \text{ and } (S_k(e, n_1, n_2, \ldots, n_k, t))_1 > \text{len}(e).$$

The set *Prog* and the relation $>$ are both primitive recursive and we saw in Problem 1.10 of *Unit 2* that the conjunction of two primitive recursive relations is also primitive recursive. Hence (ii) is a primitive recursive relation.

(iii) In the preamble we saw that the number of the instruction to be carried out at stage t is $(S_k(e, n_1, n_2, \ldots, n_k, t))_1$ and that the code number of this instruction is $(e)_a$ where $a = (S_k(e, n_1, n_2, \ldots, n_k, t))_1$. We saw in the proof of Theorem 3.9 of *Unit 2* that the set *Zinstr* of code numbers of Zero instructions is primitive recursive. Now relation (iii) can be expressed as

$$e \in \mathit{Prog} \text{ and } a \leq \text{len}(e) \text{ and } (e)_a \in \mathit{Zinstr}$$

where $a = (S_k(e, n_1, n_2, \ldots, n_k, t))_1$. From this we can deduce that it is a primitive recursive relation.

Cases (iv), (v) and (vi) can be treated in a similar way to case (iii), replacing *Zinstr* by the primitive recursive sets *Sinstr*, *Cinstr* and *Jinstr* of code numbers of Successor, Copy and Jump instructions respectively. Thus relations (i)–(vi) are all primitive recursive.

Now we turn our attention to showing how, in each of these cases, the value of $S_k(e, n_1, n_2, \ldots, n_k, t+1)$ can be obtained from $e, n_1, n_2, \ldots, n_k, t$ and $S_k(e, n_1, n_2, \ldots, n_k, t)$ using known primitive recursive functions. We give the full details for cases (i), (ii) and (iv).

(i) If e does not code a URM program then the situation number is always zero and we have

$$S_k(e, n_1, n_2, \ldots, n_k, t+1) = 0.$$

(ii) We have adopted the convention that, once the computation has halted, the value of S_k does not change. So if the computation halts at stage t or earlier then

$$S_k(e, n_1, n_2, \ldots, n_k, t+1) = S_k(e, n_1, n_2, \ldots, n_k, t).$$

(iv) Suppose that e codes a URM program and the instruction to be carried out at stage t is a Successor instruction. From the preamble, we know that the code number c of the instruction to be performed at stage t is given by $c = (e)_a$ where $a = (S_k(e, n_1, n_2, \ldots, n_k, t))_1$. Now suppose that the instruction is $S(n)$ so that its code number is $6n$. Then $c = 6n$ so $n = \text{quot}(c, 6)$, where quot is primitive recursive (*Unit 2*, Example 1.10). The effect of this instruction is to add 1 to the number in register R_n. This corresponds to increasing the exponent of p_{n+1} in the situation number by 1 and this is achieved by multiplying $S_k(e, n_1, n_2, \ldots, n_k, t)$ by p_{n+1}, where the function p which enumerates the prime numbers is

primitive recursive (*Unit 2*, Theorem 3.6). Also, since the instruction to be carried out at stage t is a Successor instruction, the instruction number at stage $t+1$ is $a+1$, so the factor 2^a in the situation number at stage t is replaced by 2^{a+1} in the situation number at stage $t+1$. Thus we have

$$S_k(e, n_1, n_2, \ldots, n_k, t+1) = 2 \times p_{n+1} \times S_k(e, n_1, n_2, \ldots, n_k, t)$$

where $n = \text{quot}(c, 6)$, $c = (e)_a$ and $a = (S_k(e, n_1, n_2, \ldots, n_k, t))_1$. This is the required expression for $S_k(e, n_1, n_2, \ldots, n_k, t+1)$ obtained by substitution from known primitive recursive functions.

The proofs for cases (iii), (v) and (vi) are similar to that for case (iv) but the details are a little more complicated and so we omit them. Thus, in each case, the value of $S_k(e, n_1, n_2, \ldots, n_k, t+1)$ can be obtained from $e, n_1, n_2, \ldots, n_k, t$ and $S_k(e, n_1, n_2, \ldots, n_k, t)$ using known primitive recursive functions.

We invite you to provide the proof for case (vi) in Additional Exercise 7 for this section.

Therefore, by Theorem 1.5 of *Unit 2*, we can conclude that the function defined by cases giving the values of $S_k(e, n_1, n_2, \ldots, n_k, t+1)$ is primitive recursive.

Hence, S_k is defined by primitive recursion from functions known to be primitive recursive. It follows that S_k is a primitive recursive function for all $k \geq 1$. ■

Theorem 2.5 is of the greatest importance. It is the key to the proof that all URM-computable functions are recursive.

2.3 URM-computable functions are recursive

Theorem 2.5 tells us that there is a primitive recursive function which describes each given URM computation. To extract from this the values of the function we are computing, all we need to do is search for a stage at which the computation has halted and then extract the output, which is the number in register R_1. The search for a stage at which the computation has halted can, as we have already hinted, be carried out by minimization. So we have now assembled all the ingredients we need for a proof of the following theorem.

> ***Theorem 2.6***
>
> Every URM-computable function is recursive.

Proof

Let $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ be a URM-computable function. Then, by hypothesis, there is a URM program which computes f. Let P be the URM program with smallest code number that computes f and let e be the code number of P. We consider the function $g : \mathbb{N}^k \longrightarrow \mathbb{N}$ given by

Using earlier notation, f is the function f_P^k and the code number e is $\gamma(P)$.

$$g(n_1, n_2, \ldots, n_k) \approx \mu t \left((S_k(e, n_1, n_2, \ldots, n_k, t))_1 > \text{len}(e) \right). \qquad (2.3)$$

Recall that S_k is the primitive recursive function such that $S_k(e, n_1, n_2, \ldots, n_k, t)$ is the situation number at stage t of the computation using the program P with input $(n_1, n_2, \ldots, n_k)$. The number $(S_k(e, n_1, n_2, \ldots, n_k, t))_1$ is that of the instruction to be carried out at stage t and $\text{len}(e)$ is the number of instructions in the program P. Thus the inequality

$$(S_k(e, n_1, n_2, \ldots, n_k, t))_1 > \text{len}(e)$$

expresses the fact that at stage t the computation has halted. So the value of $g(n_1, n_2, \ldots, n_k)$ is the number of the first stage at which the computation has halted, if there is one, and is undefined otherwise. Furthermore, since the functions involved in the inequality are all primitive recursive and the relation $>$ is primitive recursive, it follows that the inequality expresses a primitive recursive relation on $e, n_1, n_2, \ldots, n_k, t$. Thus g is a recursive function, since it is obtained by minimization on a recursive relation.

It follows that the function $h : \mathbb{N}^k \longrightarrow \mathbb{N}$ given by

$$h(n_1, n_2, \ldots, n_k) \approx S_k(e, n_1, n_2, \ldots, n_k, g(n_1, n_2, \ldots, n_k)) \tag{2.4}$$

is recursive, since it is obtained from known recursive functions using substitution. Now $h(n_1, n_2, \ldots, n_k)$ is defined if and only if the computation halts and gives the value of the situation number when the computation has halted. The output of the computation, which gives the value of f, is then the number in register R_1. But the number in register R_1 is the exponent of $p_2 = 3$ in the expression of the situation number $h(n_1, n_2, \ldots, n_k)$ in the form $p_1^a p_2^{r_1} p_3^{r_2} \ldots p_k^{r_k}$. Thus the function f is given by

$$f(n_1, n_2, \ldots, n_k) \approx (h(n_1, n_2, \ldots, n_k))_2 \,. \tag{2.5}$$

It follows that f is a recursive function, since it is obtained by substitution from known recursive functions. ■

We shall have a good deal to say about the consequences of this theorem. We note first that if we put equations (2.3), (2.4) and (2.5) together, we obtain the following formula for the values of f

$$f(n_1, n_2, \ldots, n_k) \approx (S_k(e, n_1, n_2, \ldots, n_k, \mu t\,((S_k(e, n_1, n_2, \ldots, n_k, t))_1 > \mathrm{len}(e))))_2 \,.$$

Next note the following remarkable fact. This formula works for every recursive function of k variables. The only thing that changes as we vary the function f is the number e. We say that the formula gives a *normal form* for recursive functions.

It is possible to simplify the formula in a certain way which we now explain. As it is, the value of $g(n_1, n_2, \ldots, n_k)$ which is obtained by minimization has to be substituted back into the function S_k to obtain the situation number from which we extract the output value. We can simplify this if, instead of simply searching for a number which gives a stage by which the computation has halted, we search instead for a number which encodes not only the stage number at which the computation halts but also the output number. This idea leads to a particularly simple normal form for recursive functions which we state in Theorem 2.7, which is called *Kleene's Normal Form Theorem.*

Preamble to Theorem 2.7
Suppose that e codes a URM program P which computes a function f and that the computation using the program P with input $(n_1, n_2, \ldots, n_k)$ halts at stage t_0 with output q so that $f(n_1, n_2, \ldots, n_k) = q$. Then, since the computation halts at stage t_0,

Again, e is $\gamma(P)$ and f is the function f_P^k.

$$(S_k(e, n_1, n_2, \ldots, n_k, t_0))_1 > \mathrm{len}(e)$$

and since the output is q

$$(S_k(e, n_1, n_2, \ldots, n_k, t_0))_2 = q.$$

If we put $z_0 = 2^q 3^{t_0}$ then $(z_0)_2 = t_0$ and $(z_0)_1 = q$. Thus

$$(S_k(e, n_1, n_2, \ldots, n_k, (z_0)_2))_1 > \mathrm{len}(e)$$

and

$$(z_0)_1 = (S_k(e, n_1, n_2, \ldots, n_k, (z_0)_2))_2 \,.$$

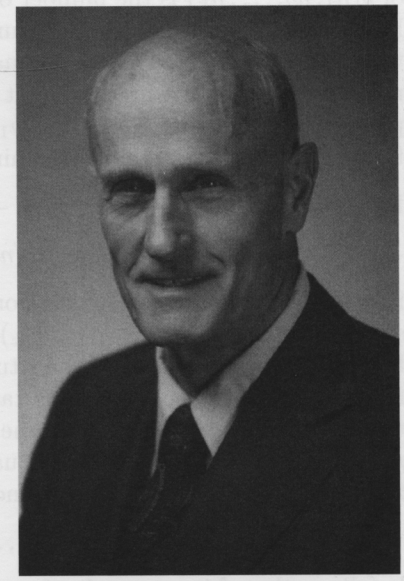
Moreover z_0 is the smallest integer with these properties and $f(n_1, n_2, \ldots, n_k) = q = (z_0)_1$.

That z_0 is the smallest such integer follows from $(z_0)_2 = t_0$ being the stage at which the computation halts.

Theorem 2.7 Kleene's Normal Form Theorem

For each integer $k \geq 1$, there is a primitive recursive $(k+2)$-place relation T_k and a primitive recursive function $U : \mathbb{N} \longrightarrow \mathbb{N}$ such that a partial function $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ is recursive if and only if, for some natural number e and all $(n_1, n_2, \ldots, n_k) \in \mathbb{N}^k$,

$$f(n_1, n_2, \ldots, n_k) \approx U(\mu z\, T_k(e, n_1, n_2, \ldots, n_k, z)). \tag{2.6}$$



The American mathematician Stephen Kleene (1909–1994) was one of the great pioneers of the theory of recursive functions. (Photo courtesy of the University of Wisconsin–Madison Archives)

Proof

Motivated by the preamble to the theorem, we let T_k be the relation defined as follows:

$$T_k(e, n_1, n_2, \ldots, n_k, z) \iff (S_k(e, n_1, n_2, \ldots, n_k, (z)_2))_1 > \text{len}(e) \text{ and } (z)_1 = (S_k(e, n_1, n_2, \ldots, n_k, (z)_2))_2$$

Then T_k is a primitive recursive relation. Also we take $U : \mathbb{N} \longrightarrow \mathbb{N}$ to be the primitive recursive function given by $U(z) = (z)_1$.

Let f be a recursive function. f is URM-computable (by Theorem 2.4) and hence there is some URM program which computes f. Let P be the URM program with smallest code number that computes f and let e be the code number of P.

Suppose that $f(n_1, n_2, \ldots, n_k)$ is defined. Then, from the preamble to the theorem, we can deduce that $\mu z\, T_k(e, n_1, n_2, \ldots n_k, z)$ is defined, and if $z_0 = \mu z\, T_k(e, n_1, n_2, \ldots, n_k, z)$ then $(z_0)_1 = f(n_1, n_2, \ldots, n_k)$, that is $U(z_0) = f(n_1, n_2, \ldots, n_k)$. Hence

$$f(n_1, n_2, \ldots, n_k) = U(\mu z\, T_k(e, n_1, n_2, \ldots, n_k, z))$$

as required.

On the other hand, if $f(n_1, n_2, \ldots, n_k)$ is undefined then the computation using the program P with input $(n_1, n_2, \ldots n_k)$ does not halt. Hence there is no z such that $T_k(e, n_1, n_2, \ldots, n_k, z)$ and so $U(\mu z\, T_k(e, n_1, n_2, \ldots, n_k, z))$ is undefined.

Thus

$$f(n_1, n_2, \ldots, n_k) \approx U(\mu z\, T_k(e, n_1, n_2, \ldots, n_k, z)).$$

Conversely a partial function obtained in this way, by substitution and minimization from a primitive recursive function U and a primitive recursive relation T_k, is recursive. ■

Kleene's Normal Form Theorem has three immediate and rather surprising consequences.

First, recall that the definition of a recursive function may involve any finite number of minimizations, and that we have allowed minimizations applied to partial functions. Also, remember that $\mu y\, R(n_1, n_2, \ldots, n_k, y)$ is equivalent to $\mu y\, (\overline{\text{sg}}(\chi_R(n_1, n_2, \ldots, n_k, y)) = 0)$, where $\overline{\text{sg}} \circ \chi_R$ is a total function. Now consider formula (2.6) in the statement of Kleene's Normal Form Theorem, which we can rewrite as

$$f(n_1, n_2, \ldots, n_k) \approx U(\mu z\ \overline{\text{sg}}(\chi_{T_k}(e, n_1, n_2, \ldots, n_k, z)) = 0).$$

The relation T_k is primitive recursive, and so therefore is its characteristic function χ_{T_k}. We know from *Unit 2* that $\overline{\text{sg}}$ is primitive recursive. Also the function U is primitive recursive. Hence χ_{T_k}, $\overline{\text{sg}}$ and U can be defined without using minimization. Thus the only minimization involved in obtaining the values of f from (2.6) is the one explicitly exhibited in formula (2.6). So we have proved the following theorem.

Unit 2, Example 1.6.

Theorem 2.8

Each recursive function can be obtained from the basic primitive recursive functions using substitution, primitive recursion and at most one minimization on a function. Moreover the minimization is applied to a total function.

The second surprising consequence of Kleene's Normal Form Theorem comes from reinterpreting it as being about URM-computable functions. This is legitimate since we know that URM-computable functions are recursive and vice versa (Theorems 2.4 and 2.6). For consider the function $\Phi_k : \mathbb{N}^{k+1} \longrightarrow \mathbb{N}$ given by

$$\Phi_k(e, n_1, n_2, \ldots, n_k) \approx U(\mu z\, T_k(e, n_1, n_2, \ldots, n_k, z))$$

where T_k and U are as in Kleene's Normal Form Theorem. Then this theorem can be reformulated in terms of URM-computable functions as follows.

Theorem 2.9 Universal URM-computable functions

For each integer $k \geq 1$, there is a URM-computable function $\Phi_k : \mathbb{N}^{k+1} \longrightarrow \mathbb{N}$ such that for each URM-computable function $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ there is a natural number e such that, for all $(n_1, n_2, \ldots, n_k) \in \mathbb{N}^k$,

$$f(n_1, n_2, \ldots, n_k) \approx \Phi_k(e, n_1, n_2, \ldots, n_k).$$

We say that the function Φ_k given by this theorem is *universal for URM-computable functions of k variables* because, as the theorem states, given Φ_k we can obtain all URM-computable functions of k variables by allowing the value of the first variable of Φ_k to vary through the natural numbers. One can think of a corresponding URM program P computing Φ_k as giving us a *universal machine* computing all URM-computable functions of k variables.

The work of this and the previous unit actually gives us a (complicated!) recipe for constructing a suitable URM program P for this universal machine. Theorem 2.5 plays a central role in this.

In Subsection 1.1 we gave a diagonal argument which showed that there are more URM-computable functions than just those that are primitive recursive. This argument hinged on the use of a function of two variables $\Phi : \mathbb{N}^2 \longrightarrow \mathbb{N}$ such that, for all $m, n \in \mathbb{N}$,

$$\Phi(m, n) = \begin{cases} f(n), & \text{if } m \text{ codes a definition of the primitive} \\ & \text{recursive function } f : \mathbb{N} \longrightarrow \mathbb{N}, \\ 0, & \text{otherwise.} \end{cases}$$

As this looks very similar to the definition of the universal function Φ_1, you might think that a similar diagonal argument ought to be applicable here. However, if such a diagonal argument did work, it would imply that there are URM-computable functions which are not recursive, contradicting Theorem 2.6. We shall now show why such an attempt at a diagonal argument fails in this case, where we attempt to argue as in Subsection 1.1.

By Theorem 2.9, we have a URM-computable function $\Phi_1 : \mathbb{N}^2 \longrightarrow \mathbb{N}$ such that for each URM-computable function $f : \mathbb{N} \longrightarrow \mathbb{N}$ there is a natural number e such that, for all $n \in \mathbb{N}$,

$$f(n) \approx \Phi_1(e, n).$$

A similar argument applies for each Φ_k, where k is a positive integer.

Now suppose the function g is obtained from Φ_1 by putting

$$g(n) \approx \Phi_1(n, n) + 1. \qquad (2.7)$$

Then g is URM-computable and so there is a natural number e_0 such that, for all $n \in \mathbb{N}$,

$$g(n) \approx \Phi_1(e_0, n).$$

Hence

$$g(e_0) \approx \Phi_1(e_0, e_0) \tag{2.8}$$

whereas from (2.7)

$$g(e_0) \approx \Phi_1(e_0, e_0) + 1. \tag{2.9}$$

From (2.8) and (2.9) we obtain

$$\Phi_1(e_0, e_0) \approx \Phi_1(e_0, e_0) + 1. \tag{2.10}$$

At first sight, we seem to have obtained the standard sort of contradiction that arises from a diagonal argument. However, Φ_1 is a partial function, so all that (2.10) tells us is that *either* both sides are defined and they are equal *or* neither side is defined. The first alternative is impossible, so we are left with the harmless conclusion that $\Phi_1(e_0, e_0)$ is not defined, rather than that there is a URM-computable function which is not recursive.

Our third consequence of Kleene's Normal Form Theorem comes from its reformulation in terms of URM programs.

Theorem 2.10 Universal URM programs

For each integer $k \geq 1$, there is a URM program P_k such that, for each URM program P, there is a natural number e such that, for all $(n_1, n_2, \ldots, n_k) \in \mathbb{N}^k$, the computation using the program P_k with input $(e, n_1, n_2, \ldots, n_k)$ has the same output as the computation using the program P with input $(n_1, n_2, \ldots, n_k)$.

We say that P_k is a *universal program for URM computations with k inputs.* The significance of this theorem is that P_k acts as an all-purpose program, since, by suitable choices of the input e in register R_1, computations using this program will mimic the behaviour of any other program.

In an idealized way, this corresponds to a modern computer capable of storing and running programs with input data. At a certain level within such a computer, a program is represented in machine code, which looks like a (doubtless huge!) binary number e; likewise the input data can be regarded as an ordered k-tuple of binary numbers for an appropriate k. The computer's central processing unit will then run any program with any input data, subject to physical constraints on the storage space required for the program and data (which don't apply to a URM). This behaviour of modern computers is now a commonplace (and of course impressive) feature of modern technology. This makes the devising of the theoretical idea of a universal program or machine by Alan Turing in 1936, before the development of this technology, all the more impressive. The relationship between the modern view of what is computable and URM-computability is the subject of the next section.

3 SCOPE AND LIMITS OF COMPUTABILITY

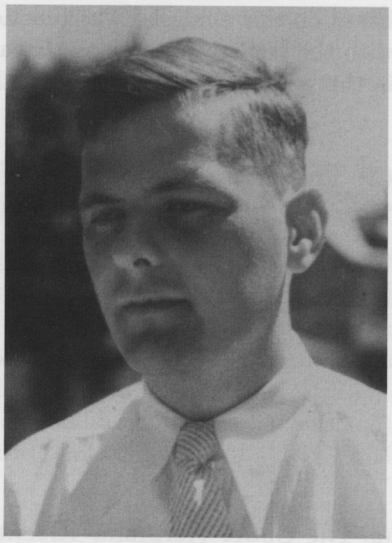
In Section 2 we saw that the set of URM-computable functions is the same as the set of recursive functions. In this section we shall see that, irrespective of how computability is defined, the set of computable functions is the same as the set of recursive functions. The key to this is *Church's Thesis*, which we discuss in Subsection 3.1. Then, in Subsection 3.2, we explore how the idea of computability relates to the concept of *decidability*.

3.1 Church's Thesis

We began the course by asking the question 'What is an algorithm?'. We chose to answer this question by looking at a seemingly simple model of computing in terms of URMs. We have now shown that URMs are much more powerful than they first appeared. However, the question now arises as to whether URM computations provide an adequate model of all algorithmic computations, or whether there are functions which we would recognize to be algorithmically computable using more powerful methods but which are not URM-computable.

This question first arose in the 1930s when recursive functions began to be studied. We give below a brief, and necessarily simplified, history of the origins of the subject. It must be remembered that, at that time, digital computers did not exist and an algorithm was thought of as a computational procedure which could be carried out mechanically by a human being. Indeed, at that time 'computer' meant a person who carried out such calculations.

The idea that the notion of an *algorithmically computable* function should be identified with that of a *recursive* function was first put forward by Alonzo Church. (In setting forward this proposal, Church used the term *effectively calculable* where we have used *algorithmically computable*.) This proposal became known as *Church's Thesis* and for the record we set it out below.



The American mathematician Alonzo Church (1903–1995) did work of major importance in logic and the theory of recursive functions. He created the lambda calculus (of which more later) which has become a valuable tool for computer science. He made his proposal in a paper which he published in 1936 but which was based on a talk that he had given to the American Mathematical Society a year earlier. (Photo © the estate of Alonzo Church)

> ***Church's Thesis***
>
> The notion of an algorithmically computable function coincides with that of a recursive function.

The first thing to note is that this thesis has the status of a philosophical theory rather than that of a mathematical theorem. We gave precise definitions of the notions of a *URM-computable function* and a *recursive function*. Because we had precise definitions, we were able to give a proof that, in fact, the two notions coincide. Church's Thesis, on the other hand, proposes to identify the precise mathematical notion of a recursive function with the intuitively understood concept of an *algorithmically computable function*. In the absence of a precise definition of the latter notion, we cannot expect to have proof of the equivalence of the two notions. The most we can hope for is to have good reasons for accepting the thesis. We present some of these reasons below. Church's Thesis is now generally accepted, but a few people might not even accept that it makes sense, as they would not allow that mathematics can deal with informal concepts of any kind. For them, mathematics *is* the study of formal systems. This view, which we call *strict formalism*, must be contrasted with the view that *ideally* mathematics should be presented as a formal system, and with the idea, which we have attributed in *Unit 1* to Hilbert, that by studying such formal systems the consistency of all, or part, of mathematics might be established by non-dubious methods of finitary reasoning.

We turn our attention now to the reasons why Church's Thesis is generally accepted, which we classify under four headings.

(1) Turing's analysis of a computation

It is anachronistic to put this reason first because Turing's work had not been published at the time at which Church stated his thesis. However, we have put it first since, from a philosophical point of view, it provides the most compelling support for Church's Thesis. Indeed, Kurt Gödel (1906–1978), the logician whose works form the focus of the remaining units of the Mathematical Logic part of this course, and who was himself influential in developing ideas about recursive functions, did not accept Church's Thesis until he saw Turing's work. The point is that Turing analysed very carefully what is involved in an algorithmic computation, and the theoretical machines which he devised, now called Turing machines, are based on this analysis. The functions computed by such machines coincide with the recursive functions. Because of the influence of Alan Turing's work in this area, the thesis is often called the *Church–Turing Thesis.*

(2) The equivalence of different characterizations of 'computable functions'

We have proved that seemingly different notions, that of a URM-computable function and that of a recursive function, turn out to be equivalent. URMs were not devised until long after Church stated his thesis. But, in the 1930s, a number of very different approaches were taken to the problem of characterizing algorithmically computable functions, and it is very striking that they all turned out to be equivalent. We list these different approaches here, with a brief description of them and a little of their history.

(a) The Gödel–Herbrand–Kleene notion of a general recursive function

Definitions of functions using substitution and recursion had been used earlier, but it was in Kurt Gödel's famous paper of 1931 that the *primitive recursive functions*, defined as in *Unit 2*, were first singled out. Gödel observed in his paper that the primitive recursive functions are computable (though this is not the term he used) — he did not introduce them for this reason, but instead because they played a technical role in his paper, as we describe in later units.

In the 1931 paper Gödel called them 'recursive functions'; the terminology changed later.

In 1934 Gödel gave a series of lectures on his work at Princeton. The notes of these lectures were written up by Stephen Kleene, whom we have already mentioned, and J. Barkley Rosser (1907–1989), who subsequently produced a strengthening of Gödel's main theorem. In these lectures Gödel introduced the idea of what he called *general recursive functions*, but which we now know simply as *recursive functions.* These were obtained from the primitive recursive functions using an idea which had been suggested to Gödel by the French logician Jacques Herbrand (1908–1931) who was tragically killed in a climbing accident when he was only 23. Essentially, Herbrand's idea was to add minimization as an additional process for generating functions. However, at this stage it was formulated in a rather different way, involving systems of equations used for deriving values of functions. It was not until Kleene published his Normal Form Theorem that it became more usual to regard this third process of minimization as we have done. Indeed, the notion of recursive functions as being generated by operations on functions (which is our approach) and as functions whose values can be derived in an equation calculus (which was the earlier approach) could be regarded as giving two distinct characterizations of the same class of functions. The proof that the two approaches lead to the same class of functions is not entirely trivial.

(b) Church's lambda calculus

Alonzo Church developed the *lambda calculus* originally as a general system for deriving theorems about numbers. His system was later shown to be inconsistent, but he was able to rescue from it the part of the system dealing with definitions of functions. The starting point is that in everyday mathematics it is hard to distinguish, for example, the function x^2 from the value of this function for an unknown x. These days, if we want to be precise we use arrow notation such as $x \longmapsto x^2$ for the function. Church's notation was different but equivalent. He used $\lambda x[x^2]$ and expressions of this type are called *lambda terms*. He introduced rules for building lambda terms and for converting one lambda term into another. For example, $\lambda x[x^2](3)$ converts into 3^2, corresponding to the fact that the squaring function applied to the number 3 produces the value 3^2. However, we emphasize that lambda terms are to be thought of as strings of symbols, rather than as numbers or functions. Numbers are represented by particular lambda terms and a function $f : \mathbb{N} \longrightarrow \mathbb{N}$ is said to be *lambda-definable* if there is a lambda term $\mathbf{f}$ such that, for all $m, n \in \mathbb{N}$, the lambda term $\mathbf{f(m)}$ converts into the lambda term $\mathbf{n}$ if and only if $f(m) = n$, where $\mathbf{m}$ and $\mathbf{n}$ are the lambda terms corresponding to the numbers m and n. With the help of Kleene, Church was able to prove that the class of lambda-definable functions coincides with the class of recursive functions. More recently the lambda calculus has proved to be useful in studying the semantics of programming languages.

lambda is the Greek letter λ.

The inconsistency caused Kleene to rewrite his own PhD thesis.

(c) Turing machines

These were described in the Appendix to *Unit 1*. Turing did his work without originally knowing about the alternative approaches mentioned above. However, he learned of this other work before publishing his paper in 1936 on what we now know as Turing machines and, at the end of the paper, he sketched a proof that his notion of a computable function is equivalent to Church's concept of a lambda-definable function. Because of its simplicity and the convincing nature of Turing's analysis of a computation, Turing's approach remains the paradigm for a formal characterization of computable functions.

(d) Post systems

Emil Post (1897–1954), independently of Turing but at about the same time, arrived at an approach to computable functions quite close to Turing's conception. Turing machines manipulate symbols on a tape. For Post, a computation consisted of manipulating strings of symbols according to specific rules called *production rules*. A *Post system* consists of a finite set of initial strings and a finite set of production rules for transforming one string of symbols into another.

For example, if we start from the initial string

$$f\,1 = 1$$

and use the production rule

$$f\,A = B \longrightarrow f\,A\,1 = B\,A\,A\,1$$

where A, B can be any strings, then we can successively derive the strings

$$f\,1\,1 = 1\,1\,1\,1$$

$$f\,1\,1\,1 = 1\,1\,1\,1\,1\,1\,1\,1$$

and so on.

If, for integers $n \geq 1$, we use $\mathbf{n}$ as an abbreviation for the string consisting of n consecutive 1s, we see that the initial string and the strings we can derive from it can be written as

$$f\,\mathbf{1} = \mathbf{1}$$

$$f\,\mathbf{2} = \mathbf{4}$$

$$f\,\mathbf{3} = \mathbf{9}$$

and so on; that is, we can derive all the strings which express correctly the values of the function $n \longmapsto n^2$. A function is said to be *Post-computable* if there is a finite set of initial strings and a finite set of rules such that the strings we can derive are precisely those expressing the correct values of the given function. Again, it is possible to show that the class of Post-computable functions coincides with the class of recursive functions.

Post systems fit in very naturally with algebraic manipulations. For example, if you are familiar with the notion of an element g of a group having order 3, you will see that this corresponds to the production rule

$$A\,g\,g\,g\,B \longrightarrow A\,B$$

for manipulating strings of symbols corresponding to group elements. Post used these methods to derive results about computability in semigroups and these ideas have been subsequently extended to groups. Production rules also play a role in transformational grammars in the style of Chomsky.

It is very striking that all these different attempts to give a characterization of the notion of a computable function have led to exactly the same class of functions. This supports the idea that they are all successful in providing a formal definition of a natural informal notion. It seems implausible that everyone has made the same mistake!

(3) The stability of these notions

This reason for accepting the truth of Church's Thesis is closely related to the previous one. In each case, attempts to vary the definition by seemingly strengthening it in some way have not led to a larger class of functions. For example, it is possible to formulate Turing machines so that they operate on a two-dimensional array of squares rather than a one-dimensional tape. However, it is possible to prove that this has no effect on which functions are computable, though it may make some computations easier.

Likewise, it is possible to vary the set of basic instructions for URMs. If the instruction set is too small, it may no longer be possible to compute all recursive functions. For example, if Successor and Jump instructions are omitted, there are very few functions that can still be computed. No one, however, has found extra acceptable instructions which increase the set of functions which then become computable.

We indicated informally in Section 1 how the diagonal method can be used to produce a URM-computable function which is not primitive recursive. In Section 2, we explained why it is not possible to apply the diagonal construction to the class of recursive functions to obtain a URM-computable function which is not recursive. Kleene has commented (in 1981) that this point assumed great significance for him.

> When Church proposed his thesis, I sat down to disprove it by diagonalizing out of the class of λ-definable functions. But, quickly realizing that the diagonalization cannot be done effectively, I became overnight a supporter of the thesis.

(4) Empirical evidence

No one has produced an example of a function which would be generally recognized as computable but which is not recursive. It might be thought that, as computers have become more powerful, and as higher-level programming languages have been developed, it has become possible to compute more and more functions; but this is not the case. From a practical point of view, more powerful computers increase the number of computations that can be carried out in a reasonable amount of time, and in many applications this is very important. But from a theoretical point of view, high-level programming languages implemented on powerful computers are not able to go beyond URM programs in relation to which functions may actually be computed. This seems less surprising when it is appreciated that high-level programs are not implemented directly, but are compiled into a lower-level language (machine code) which is at about the same level of complexity as the URM programming language.

Uses of Church's Thesis

Church's Thesis is used in two ways, one practical and the other philosophical.

The practical use of Church's Thesis is that, when we have a function which can be seen to be algorithmically computable, we can assume that this function is recursive (or, equivalently, URM-computable) without writing out a proof in detail. For example, the function reg defined by

$$\mathrm{reg}(n) = \begin{cases} \rho(P), & \text{if } n \text{ codes the URM program } P, \\ 0, & \text{otherwise,} \end{cases}$$

where $\rho(P)$ is the maximum value u such that the program P uses the register R_u, can be assumed to be recursive. We can argue this as follows. Given n, we can work out in a finite number of steps whether n codes a URM program P. If it does, then (again in a finite number of steps) we can decode n to obtain all the instructions of P and, by scanning each of these, establish the value of $\rho(P)$. Thus reg is clearly algorithmically computable. Hence, by Church's Thesis, it is recursive.

This is a very weak use of Church's Thesis. It just expresses our confidence, based on experience, that, if pressed, we could turn an informal algorithm for computing the values of the function reg into a precise proof that it is a recursive function.

In Additional Exercise 1 for this section we invite you to provide such a proof.

The philosophical use of Church's Thesis arises when we deduce general results about algorithmic computability from theorems about recursive functions. In the next subsection we give some results concerning problems that cannot be decided using recursive functions. These results gain more significance if we accept Church's Thesis because we can then interpret them as saying that the problems cannot be decided algorithmically. Some of these general results exploit the following problem.

Problem 3.1

Let $f\colon \mathbb{N}^k \longrightarrow \mathbb{N}$ and $g\colon \mathbb{N}^k \longrightarrow \mathbb{N}$ be recursive functions (not necessarily total), where $k \geq 1$, and let R be a recursive k-place relation such that

if $R(n_1, n_2, \ldots, n_k)$ holds, then $f(n_1, n_2, \ldots, n_k)$ is defined,

if $R(n_1, n_2, \ldots, n_k)$ does not hold, then $g(n_1, n_2, \ldots, n_k)$ is defined.

Let $h\colon \mathbb{N}^k \longrightarrow \mathbb{N}$ be the function defined by

$$h(n_1, n_2, \ldots, n_k) = \begin{cases} f(n_1, n_2, \ldots, n_k), & \text{if } R(n_1, n_2, \ldots, n_k) \text{ holds,} \\ g(n_1, n_2, \ldots, n_k), & \text{otherwise,} \end{cases}$$

so that, from the information given about the relation R and the functions f and g, the function h is total.

(a) Show that h is recursive (equivalently, that it is URM-computable) by outlining an informal algorithm for computing its values.

In Additional Exercise 2 for this section we invite you to give a more formal proof that h is recursive.

(b) The method of proof used for Theorems 1.3, 1.4 and 1.5 and Problem 1.15 of *Unit 2* might suggest that h can be shown to be recursive by using the equation

$$h(n_1, n_2, \ldots, n_k) = f(n_1, n_2, \ldots, n_k)\chi_R(n_1, n_2, \ldots, n_k) + g(n_1, n_2, \ldots, n_k)\,\overline{\mathrm{sg}}(\chi_R(n_1, n_2, \ldots, n_k))$$

and saying that the right-hand side is obtained by substitution from known recursive functions, so is recursive, and hence h is recursive. Unfortunately this argument is fallacious. Can you explain why?

3.2 Algorithmically undecidable problems

The first general problem about algorithmic computability that we shall consider concerns deciding whether or not a given URM program halts for a given input. Our second general problem concerns deciding whether a URM program halts for *all* inputs, i.e. whether it computes a total function.

Before looking at these problems, it is worth explaining what we mean by 'deciding' a problem or question. First we must represent the question in some formal way that can ultimately be turned into a relation between natural numbers. By saying the question is 'decidable' we mean that there is an algorithmic procedure which tells us whether or not this relation is satisfied, that is, gives an answer 'yes' or 'no'. We shall then say that the question is *algorithmically decidable*. The algorithmic procedure is often called a *decision procedure*. This will give you an idea of how what we have done in *Units 1* to *3* relates to Leibniz's Question, asking if there is an algorithm for deciding which statements of number theory are true.

Some books use *algorithmically solvable* instead of algorithmically decidable.

For the program we gave at the start of Subsection 1.2, it was easy to see that the computation halts whenever the input number n is even but does not halt when n is odd. In more complicated cases it can be much harder to decide whether a computation halts for a particular input. Consider, for example, the following URM program P.

This was the program computing the function $f\colon \mathbb{N} \longrightarrow \mathbb{N}$ given by

$$f(n) = \begin{cases} \frac{1}{2}n, & \text{if } n \text{ is even,} \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

1 $S(2)$
2 $J(1,2,23)$
3 $J(1,3,9)$
4 $S(3)$
5 $J(1,3,13)$
6 $S(3)$
7 $S(4)$
8 $J(1,1,3)$
9 $C(4,1)$
10 $Z(3)$
11 $Z(4)$
12 $J(1,1,2)$
13 $Z(4)$
14 $J(3,4,19)$
15 $S(1)$
16 $S(1)$
17 $S(4)$
18 $J(1,1,14)$
19 $S(1)$
20 $Z(3)$
21 $Z(4)$
22 $J(1,1,3)$

This program P carries out what is known as the *Collatz process.*

Note that the program doesn't halt for input 0. With input n, the computation replaces n by $\frac{1}{2}n$ if n is even and by $3n+1$ if n is odd and $n \neq 1$. This process is repeated unless and until the number 1 is reached, in which case the computation halts. For example, with input 5 the numbers that are successively calculated are

$$5 \to 16 \to 8 \to 4 \to 2 \to 1$$

and with input 7 they are

$$7 \to 22 \to 11 \to 34 \to 17 \to 52 \to 26 \to 13 \to$$
$$40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1.$$

An equivalent open question is whether the function f_P^1 is the constant function C_1.

In both cases the computation halts with output 1. It is conjectured that this process ends up with the number 1 whatever the input. This has not yet, however, been proved. So at the moment it is not known whether the computation using this program halts for every input.

The problem of algorithmically deciding whether or not a computation using a particular URM program halts for a particular input is called the *Halting Problem.* We let $H\colon \mathbb{N}^2 \longrightarrow \mathbb{N}$ be the function defined by

$$H(m,n) = \begin{cases} 1, & \text{if } m \text{ codes a URM program which} \\ & \text{halts with input } n, \\ 0, & \text{otherwise.} \end{cases}$$

If the function H were algorithmically computable then the Halting Problem would be algorithmically decidable. However, as you will suspect from the previous discussion, we can prove that this is not the case.

We can give the argument in terms of either URM programs or recursive functions. We choose the latter.

Suppose that H is recursive. We shall show that this leads to a contradiction.

Recall that Φ_1 is the universal URM-computable function for URM-computable functions of one variable given by Theorem 2.9. Now let $f\colon \mathbb{N} \longrightarrow \mathbb{N}$ be the function defined by

$$f(n) = \begin{cases} \Phi_1(n,n), & \text{if } H(n,n) = 1, \\ 0, & \text{otherwise.} \end{cases}$$

As H is recursive, the relation $H(n,n) = 1$ is recursive. The universal function Φ_1 is recursive and $\Phi_1(n,n)$ is defined when $H(n,n) = 1$.

The constant function zero is recursive and always defined. So the function f is total and, by the result of Problem 3.1, is also recursive.

It follows immediately that the function $g\colon \mathbb{N} \longrightarrow \mathbb{N}$ given by

$$g(n) = f(n) + 1$$

is also total and recursive. As g is recursive, Theorem 2.9 tells us that there is some natural number e such that, for all n, $g(n) = \Phi_1(e, n)$, that is, the URM program coded by e computes g. Hence

$$g(e) = \Phi_1(e, e). \tag{3.1}$$

But since e codes a URM program which halts with input e, $H(e, e) = 1$ and $f(e) = \Phi_1(e, e)$. Therefore, from the definition of g,

$$g(e) = \Phi_1(e, e) + 1. \tag{3.2}$$

Equations (3.1) and (3.2) are contradictory. This contradiction has arisen because we made the assumption that the function H is recursive. Hence this assumption must be wrong and so we have proved the following.

Note that we write $=$ here rather than $\approx$ as we know that f is a total function.

Another fine use of a diagonal argument!

Theorem 3.1 The Halting Problem is not algorithmically decidable

The function $H\colon \mathbb{N}^2 \longrightarrow \mathbb{N}$ given by

$$H(m, n) = \begin{cases} 1, & \text{if } m \text{ codes a URM program which} \\ & \text{halts with input } n, \\ 0, & \text{otherwise,} \end{cases}$$

is *not* recursive.

The algorithmic undecidability of the Halting Problem is significant for computer programming. A computation which goes into some form of loop which it repeats over and over again does not halt. It would be good to be able to write computer programs with the certainty that bugs of this kind will not occur. However, the theorem we have just proved shows that there is no general algorithm for checking computer programs. Of course, for *particular* programs it may be possible to decide algorithmically for which inputs a computation will halt. The significance of our theorem is that there is no *general* algorithm for deciding whether a program with a particular input will halt or not. In the light of this, given the many and important uses of computer programs, often of considerable complexity, a great deal of effort and ingenuity goes into writing programs in ways that maximize the chances that their correctness can be checked.

A related problem is to identify URM programs which compute total functions. For simplicity, we shall confine ourselves to those programs which compute a total function of one variable. Let *Tot* be the set of numbers that code such programs. We are going to show that *Tot* is not a recursive set, so that its characteristic function χ_{Tot} is not recursive and hence that there is no way of deciding algorithmically whether an arbitrary URM-computable function $f\colon \mathbb{N} \longrightarrow \mathbb{N}$ is total.

Again the idea of the proof is quite simple. We shall show that if *Tot* is recursive then we can find a URM-computable function $h : \mathbb{N} \longrightarrow \mathbb{N}$ which enumerates the members of *Tot*, that is, a one–one function h such that

$$Tot = \{h(0), h(1), h(2), \ldots\}.$$

Then we can define a total URM-computable function $\Phi : \mathbb{N}^2 \longrightarrow \mathbb{N}$ by putting

$$\Phi(m, n) = \Phi_1(h(m), n),$$

meaning that $\Phi(m, n) = f(n)$ where f is the total function of one variable computed by the program with code number $h(m)$. A diagonal argument

applied to Φ will produce a contradiction from which we can conclude that *Tot* is not recursive.

> ***Theorem 3.2***
>
> The set *Tot* of numbers that code URM programs which compute a total function of one variable is *not* recursive.

Proof

We assume, to obtain a contradiction, that *Tot* is a recursive set.

First we define a recursive function h which enumerates the code numbers of URM programs which compute a total function of one variable. The program of this sort with the smallest code number is

$$1 \quad Z(1)$$

which computes the zero function; its code number is $2^3 = 8$. So we shall define the recursive function h by

$$h(0) = 8,$$
$$h(n+1) = \mu z\,(z \in Tot \text{ and } h(n) < z).$$

As we are supposing that the set *Tot* is recursive, we can deduce from its definition that h is recursive.

As there are infinitely many URM programs computing a total function of one variable, $h(n)$ is always defined. More importantly, if $f : \mathbb{N} \longrightarrow \mathbb{N}$ is a total recursive function, there is some number m such that the URM program with code number $h(m)$ computes f.

For instance, for each integer k, the constant function $C_k : n \longmapsto k$ is URM-computable and total.

Now let $\Phi : \mathbb{N}^2 \longrightarrow \mathbb{N}$ be the function defined by

$$\Phi(m, n) = \Phi_1(h(m), n),$$

where Φ_1 is the universal function defined in Theorem 2.9. Then Φ is a total recursive function (since Φ_1 and h are recursive and since the program coded by $h(m)$ halts for all inputs n) with the property that if $f : \mathbb{N} \longrightarrow \mathbb{N}$ is a total recursive function then there is some number m such that $\Phi(m, n) = f(n)$ for all $n \in \mathbb{N}$.

We can use $=$ rather than $\approx$ as the right-hand side is defined for all m and n, even though Φ_1 is not a total function.

Now we can obtain a contradiction using a diagonal argument. The details are left to you (see Problem 3.2). We deduce that the set *Tot* is not recursive. ■

This result can be restated as: the problem of determining whether a given URM program P computes a total function f_P^1, that is, halts for all inputs in register R_1 when 0 is input in all other registers, is algorithmically undecidable.

Problem 3.2

Complete the proof of Theorem 3.2.

The result that the set *Tot* is not recursive can be used to show that other sets are not recursive. We shall give one example concerning URM programs which compute a given URM-computable function g. To lead into this example, we ask you to show that there are infinitely many URM programs which compute any given g.

Problem 3.3

Suppose that the function $g : \mathbb{N} \longrightarrow \mathbb{N}$ is computable by the URM program Q. Show that there are infinitely many URM programs which compute g. *Hint*: Show how to concatenate other instructions with Q to produce programs which have the same effect on all inputs as Q.

So g is the function f_Q^1.

Our solution to Problem 3.3 describes infinitely many URM programs computing g, but by no means all of the programs which do this — there will be many that bear no obvious relation or resemblance to the program Q. The sets of all URM programs and of those which compute a specific function g are thus so big and complicated that one might suspect that deciding algorithmically whether a specific URM program P computes g will be impossible.

Of course, in practical computing, one strives to design a program to compute g in ways that enable one to verify that it achieves this goal.

Example 3.1

Let X be the set of numbers which code URM programs which compute the function $g\colon \mathbb{N} \longrightarrow \mathbb{N}$ given by $g(n) = n^2$. We shall show that the set X is not recursive.

Our strategy is to give an informal algorithmic procedure for turning any URM program P coded by a number e into a program P^* such that:

> if the function f_P^1 of one variable computed by P is total, then P^* will compute the function g;
>
> if f_P^1 is not total, then P^* will not compute g.

Once we have such a procedure, we suppose that X is recursive, so that there is an algorithm for testing whether a number m is in X. We then test whether a number e is in *Tot* as follows. Given e, check whether or not it codes a URM program. If it does not code a program, e is not in *Tot*. If it does code a program, P say, recover the instructions of P from e and use them to turn P into the corresponding program P^*. Work out the code number $\gamma(P^*)$ of P^* and test whether it is in the set X. If it is, then P computes a total function f_P^1, so e is in *Tot*; and if it is not, then f_P^1 is not total and e is not in *Tot*. So we have an algorithm for testing whether e is in *Tot*, so that the set *Tot* is recursive, which contradicts Theorem 3.2. We conclude that there is no algorithm for testing whether m is in X, so that X is not recursive.

If you would like a taste of part of the process of converting e into the code number of P^*, you might like to try Additional Exercise 4 for this section.

To create P^* from P, we shall exploit one of the URM programs, Q say, which computes g.

The function $g(n) = n^2$ is URM-computable. See the program P on page 20 of *Unit 1*.

Suppose that we are given a number e and have already determined that it codes a URM program P. We can also compute the maximum register number $\rho(P)$ used in P, namely $\mathrm{reg}(e)$, and then the maximum of $\rho(P)$ and $\rho(Q)$. We then construct P^* from P by concatenating programs and extra instructions as follows. First we have the instruction

$$C(1, \max(\rho(P), \rho(Q)) + 1)$$

This has the effect of copying an input n to P^* into a register not used in P or Q.

Next we concatenate the program

$$P$$

This has the effect of computing $f_P^1(n)$, which might of course not be defined for some n if P fails to halt with this input, in which case P^* also does not halt with this input.

Our next instructions would, in the case when $f_P^1(n)$ is defined, clear the other registers used in Q and transfer the original input n back into register R_1, as follows:

$$\begin{array}{l} Z(1) \\ Z(2) \\ \vdots \\ Z(\rho(Q)) \\ C(\max(\rho(P), \rho(Q)) + 1,\ 1) \end{array}$$

The $Z(1)$ instruction is redundant. We include it only to clarify what is being done.

Finally we concatenate the above program with

Q

which has the effect that the whole concatenated program will compute n^2 exactly when $f_P^1(n)$ is defined.

To summarize, P^* is the concatenated program

$C(1, \max(\rho(P), \rho(Q)) + 1)$
P
$Z(1)$
$Z(2)$
$\vdots$
$Z(\rho(Q))$
$C(\max(\rho(P), \rho(Q)) + 1, 1)$
Q

With input n, this program P^* outputs n^2 exactly when $f_P^1(n)$ is defined, so that P^* computes the function g exactly when f_P^1 is a total function, as required. ♦

The result of Example 3.1 can be restated as: the problem of determining whether a URM program computes the squaring function of one variable is algorithmically undecidable.

Problem 3.4

(a) Give a URM program Q which computes the zero function given by $\text{zero}(n) = 0$ for all $n \in \mathbb{N}$.

(b) Give an informal algorithmic procedure for turning any URM program P coded by a number e into a program P^* such that:

if the function f_P^1 of one variable computed by P is total, then P^* will compute the zero function;

if f_P^1 is not total, then P^* will not compute the zero function.

(c) Explain why the set *Zero* of code numbers of URM programs which compute the zero function is not recursive.

You met the set *Zero* in *Unit 2*.

The result of Problem 3.4 can be restated as: the problem of determining whether a URM program computes the zero function is algorithmically undecidable.

Problem 3.5

For each $e \in \mathbb{N}$ which codes a URM program, let W_e be the set

$$W_e = \{k : \Phi_1(e, k) \text{ is defined}\},$$

where Φ_1 is the universal function defined in Theorem 2.9. If e does not code a URM program, let W_e be the empty set. So, if e codes a URM program, W_e is the domain of the function $f : \mathbb{N} \longrightarrow \mathbb{N}$ coded by e. Let D be the two-place relation given by

$$D(n, m) \iff W_n = W_m.$$

(a) Show that if $D(e, 8)$ holds then the function $f : \mathbb{N} \longrightarrow \mathbb{N}$ coded by e is total. *Hint*: For the significance of the number 8, see the proof of Theorem 3.2.

(b) Deduce that the relation D is not recursive.

The result of Problem 3.5 can be restated as: the problem of determining whether two URM-computable functions have the same domain is algorithmically undecidable.

There are many results showing that various questions about computer programs are algorithmically undecidable. Some occur in the Additional Exercises on this section. For more examples, including algorithmically undecidable problems in areas of mathematics other than computer programming, you should look at some of the books listed in the Suggestions for Further Reading for this part of the course at the end of *Unit 8*. A major example is the algorithmic decidability of the truth of statements of number theory, which is what we shall look at in the remaining units of the Mathematical Logic part of the course.

SUMMARY

In this unit we have broadened the definition of URM computability so that each URM program P computes a function f_P^k of k variables, for each $k \geq 1$, where f_P^k is, in general, a partial function, that is, its domain is a subset of $\mathbb{N}^k$. The computation using the program P with input $(n_1, n_2, \ldots, n_k)$ halts with output $f_P^k(n_1, n_2, \ldots, n_k)$ if and only if $(n_1, n_2, \ldots, n_k)$ is in the domain of f_P^k.

Next, motivated by the search for the stage at which a computation halts, we introduced the operation of minimization and saw that minimization, even when applied to a total function, will in general produce a partial function. We found that minimization applied to a URM-computable function or relation produces a URM-computable function. This led us to introduce the class of recursive functions as those functions that can be obtained from the basic primitive recursive functions using the operations of substitution, primitive recursion and minimization. We found that every recursive function is URM-computable.

By a careful study of the coding of URM computations, we were able to show that the converse is also true, that is, every URM-computable function is recursive. A spin-off was Kleene's Normal Form Theorem, from which we were able to establish the existence of universal URM-computable functions and universal URM programs.

The coincidence of the classes of URM-computable and recursive functions is evidence for Church's Thesis that the informal and intuitive idea of an algorithmically computable function is captured by the notion of a recursive function. We gave a brief summary of the history of Church's Thesis and of the compelling evidence for its acceptance. We noted the philosophical and practical consequences of Church's Thesis.

Finally we looked at some problems about URM-computability, such as the Halting Problem, which are not algorithmically decidable. We noted that algorithmically undecidable problems arise and are significant in other areas of mathematics as well as in the theory of computer programs.

In the remainder of the Mathematical Logic units, armed with our description of algorithmic computability, we shall turn to the remaining machinery required to resolve Leibniz's and Hilbert's Questions, namely the study of mathematical logic. We begin by introducing a formal language for number theory. The notion of algorithmic computability will be present in the rest of the course, but in the background for some time. Recursive functions will play an important role in the proofs of the major theorems of the course in *Units 7* and *8*.

OBJECTIVES

We list those topics on which we may set assessment questions to test your understanding of this unit.

After working through the unit you should be able to:

(a) calculate values of functions defined by minimization and, in simple cases, determine whether such functions are defined for certain inputs;

(b) give proofs that functions are recursive using minimization;

(c) given a URM program which computes a particular function, determine a URM program which computes the function derived from it by minimization;

(d) calculate the situation number corresponding to a given stage of a given URM computation;

(e) given a number, determine the situation, if any, which it codes;

(f) exploit Church's Thesis to show that functions, relations and sets are recursive;

(g) show that certain functions, sets and relations are not recursive.

ADDITIONAL EXERCISES

Most of these exercises provide further practice, should you feel you need it, in handling the main ideas in the unit on which you are likely to be assessed.

There are a few harder problems, labelled as such in the margin. These are harder than any of the problems you are likely to encounter in the assessment and are included solely as challenges for the interested student.

Section 1

1 Show that the partial function $f : \mathbb{N} \longrightarrow \mathbb{N}$ given by

$$f(n) = \begin{cases} 1, & \text{if } n > 0, \\ \text{undefined}, & \text{otherwise}, \end{cases}$$

is URM-computable.

2 Let P be the following URM program.

1 $J(1,3,5)$
2 $S(2)$
3 $S(3)$
4 $J(1,1,1)$
5 $J(1,2,5)$

Describe the function f_P^1 of one variable, the function f_P^2 of two variables and the function f_P^3 of three variables computed by P.

Section 2

1 Show that the partial function $f : \mathbb{N} \longrightarrow \mathbb{N}$ given by

$$f(n) = \begin{cases} 1, & \text{if } n > 0, \\ \text{undefined}, & \text{otherwise}, \end{cases}$$

is recursive.

2 The partial function $g : \mathbb{N}^2 \longrightarrow \mathbb{N}$ is given by

$$g(n,m) = \begin{cases} 0, & \text{if } |n-m| < 3, \\ \text{undefined}, & \text{otherwise,} \end{cases}$$

and the partial function $f : \mathbb{N} \longrightarrow \mathbb{N}$ is given by

$$f(n) \approx \mu y\,(g(n,y) = 0).$$

Determine, for $0 \leq n \leq 5$, whether or not $f(n)$ is defined and evaluate $f(n)$ in the cases where it is defined.

3 The following URM program computes a function $g : \mathbb{N}^2 \longrightarrow \mathbb{N}$.

1 $J(2,3,6)$
2 $S(4)$
3 $S(4)$
4 $S(3)$
5 $J(1,1,1)$
6 $C(1,3)$
7 $C(4,2)$
8 $J(1,4,16)$
9 $J(2,3,14)$
10 $S(3)$
11 $S(4)$
12 $S(5)$
13 $J(1,1,8)$
14 $Z(1)$
15 $J(1,1,17)$
16 $C(5,1)$

Find a URM program which computes the partial function $f : \mathbb{N} \longrightarrow \mathbb{N}$ given by

$$f(n) \approx \mu y\,(g(n,y) = 0).$$

4 Let P be the URM program

1 $J(1,3,6)$
2 $S(2)$
3 $S(2)$
4 $S(3)$
5 $J(1,1,1)$
6 $C(2,1)$

and let e be the number which codes this program. Calculate the situation numbers $S_1(e,1,t)$ for $0 \leq t \leq 10$.

5 Specify the situation corresponding to each of the following situation numbers for a URM program which uses the registers R_1, R_2 and R_3 only.

(a) 2 (b) 90 (c) 180 (d) 196

6 Let $g : \mathbb{N}^2 \longrightarrow \mathbb{N}$ be the partial function given by putting $g(e,n) = 0$ if e does not code a URM program and $g(e,n) = t_0$ if the computation with input n of the URM program with code number e halts at stage t_0, while $g(e,n)$ is undefined otherwise. Prove that g is a recursive function. Harder problem

7 Supply the details for case (vi) in the proof of Theorem 2.5. Harder problem

Section 3

1 The function reg: $\mathbb{N} \longrightarrow \mathbb{N}$ is defined by

Harder problem

$$\text{reg}(n) = \begin{cases} \rho(P), & \text{if } n \text{ codes the URM program } P, \\ 0, & \text{otherwise,} \end{cases}$$

where $\rho(P)$ is the maximum value u such that the program P uses the register R_u. Prove that reg is a primitive recursive function. *Hint*: First use the function $\max : \mathbb{N}^2 \longrightarrow \mathbb{N}$ to show that, for any primitive recursive function f: $\mathbb{N} \longrightarrow \mathbb{N}$, the function g: $\mathbb{N} \longrightarrow \mathbb{N}$ defined by

$$g(n) = \begin{cases} 0, & \text{if } n = 0, \\ \max\{f(k) : 1 \leq k \leq n\}, & \text{if } n \geq 1, \end{cases}$$

is primitive recursive. Then show that the function M given by

$$M(n) = \begin{cases} \text{the maximum register} \\ \text{number used by } I, & \text{if } n \text{ codes a URM instruction } I, \\ 0, & \text{otherwise,} \end{cases}$$

is primitive recursive.

2 Let $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ and $g : \mathbb{N}^k \longrightarrow \mathbb{N}$ be recursive functions (not necessarily total) and let R be a recursive k-place relation such that

Harder problem

if $R(n_1, n_2, \ldots, n_k)$ holds, then $f(n_1, n_2, \ldots, n_k)$ is defined,
if $R(n_1, n_2, \ldots n_k)$ does not hold, then $g(n_1, n_2, \ldots, n_k)$ is defined.

Let $h : \mathbb{N}^k \longrightarrow \mathbb{N}$ be the total function defined by

$$h(n_1, n_2, \ldots, n_k) = \begin{cases} f(n_1, n_2, \ldots, n_k), & \text{if } R(n_1, n_2, \ldots, n_k), \\ g(n_1, n_2, \ldots, n_k), & \text{otherwise.} \end{cases}$$

Show that h is a recursive function without appealing to Church's Thesis. *Hint*: We need to avoid the pitfall pointed out in the solution to Problem 3.1(b). Our solution uses Kleene's Normal Form Theorem to obtain a single recursive relation S such that $h(n_1, n_2, \ldots, n_k) = U(\mu z\, S(n_1, n_2, \ldots, n_k, z))$. It also makes use of the results for recursive relations equivalent to the results for primitive recursive relations in Problem 1.10 of *Unit 2*. You may assume that these results hold.

3 Give an example of a two-place relation R such that the function $f : \mathbb{N}^2 \longrightarrow \mathbb{N}$ given by

$$f(m, n) = \begin{cases} 1, & \text{if } R(m, n) \text{ holds,} \\ 0, & \text{otherwise,} \end{cases}$$

is *not* recursive but the function $g : \mathbb{N}^2 \longrightarrow \mathbb{N}$ given by

$$g(m, n) = \begin{cases} 1, & \text{if } R(m, n) \text{ holds,} \\ \text{undefined}, & \text{otherwise,} \end{cases}$$

is recursive. *Hint:* Look at Theorem 3.1.

4 In this exercise, we ask you to investigate one part of how to convert the code number of a URM program P into the code number of the program P^* in Example 3.1. In this example the program P is concatenated with other instructions, and some of the jump instructions of P might thus get adjusted. The point of this exercise is to show that, in such an adjustment, the code number of the adjusted program is a primitive recursive function of the code number of P. Harder problem

We say that a URM program P with k instructions is in *standard form* if, for every Jump instruction $J(m,n,q)$ occurring in P, we have $q \leq k+1$. If P is a URM program with k instructions then the standard form of P is the URM program obtained from P by replacing each Jump instruction $J(m,n,q)$ of P where $q > k+1$ by $J(m,n,k+1)$. We denote the standard form of P by P'. Show that the function sf defined by

$$\mathrm{sf}(e) = \begin{cases} \gamma(P'), & \text{if } e \text{ codes a URM program } P, \\ 0, & \text{otherwise,} \end{cases}$$

is primitive recursive, where $\gamma(P')$ is the number which codes the URM program P'.

5 (a) Give a URM program Q which computes the successor function given by $\mathrm{succ}(n) = n+1$ for all $n \in \mathbb{N}$.

(b) Give an informal algorithmic procedure for turning any URM program P coded by a number e into a program P^* such that:

if the function f_P^1 of one variable computed by P is total, then P^* will compute the successor function;

if f_P^1 is not total, then P^* will not compute the successor function.

6 Show that the two-place relation E given by

$$E(m,n) \iff m \text{ and } n \text{ code URM programs which compute the same function of one variable}$$

is not recursive. *Hint*: Exploit the result of Problem 3.4 about the set *Zero*.

7 For each $e \in \mathbb{N}$ which codes a URM program, let W_e be the set Harder problem

$$W_e = \{k : \Phi_1(e,k) \text{ is defined}\},$$

where Φ_1 is the universal function defined in Theorem 2.9. If e does not code a URM program, let W_e be the empty set. So, if e codes a URM program, W_e is the domain of the function $f : \mathbb{N} \longrightarrow \mathbb{N}$ coded by e. Let K be the set

$$K = \{e \in \mathbb{N} : e \in W_e\}.$$

Show that K is not a recursive set.

SOLUTIONS TO THE PROBLEMS

Solution 1.1

The computation halts only if the initial contents of registers R_1 and R_2 are unequal. Thus the functions are given by:

$$f_P^1(n) = \begin{cases} n, & \text{if } n > 0, \\ \text{undefined}, & \text{if } n = 0; \end{cases}$$

$$f_P^2(n_1, n_2) = \begin{cases} n_1, & \text{if } n_1 \neq n_2, \\ \text{undefined}, & \text{if } n_1 = n_2. \end{cases}$$

As the largest register number used by the program P is 2, the output of any computation is independent of the initial contents of the registers R_i for all $i \geq 3$. The functions f_P^k for $k \geq 3$ are thus similar to the function f_P^2 and are given by

$$f_P^k(n_1, n_2, \ldots, n_k) = \begin{cases} n_1, & \text{if } n_1 \neq n_2, \\ \text{undefined}, & \text{if } n_1 = n_2. \end{cases}$$

Solution 1.2

The following URM program computes the given partial function.

1 $J(1,2,7)$
2 $S(2)$
3 $S(2)$
4 $S(2)$
5 $S(3)$
6 $J(1,1,1)$
7 $C(3,1)$

It is an obvious modification of the URM program, given near the start of Subsection 1.2, which computes the function

$$f(n) = \begin{cases} \frac{1}{2}n, & \text{if } n \text{ is even}, \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

Solution 2.1

It can be seen that, in general

$$g(n) = \begin{cases} \frac{1}{2}n, & \text{if } n \text{ is even}, \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

Thus $g(0) = 0$, $g(2) = 1$, $g(4) = 2$, $g(6) = 3$ while $g(1)$, $g(3)$, $g(5)$ are undefined.

Solution 2.2

As $g(n)$ is the smallest number y such that $n \leq y^2$, we obtain the following table of values.

n	0	1	2	3	4	5	6	7	8	9
$g(n)$	0	1	2	2	2	3	3	3	3	3

Solution 2.3

(a) We have $f(2,0) = 2$, $f(2,1) = 1$ and $f(2,y)$ is undefined if $y \geq 2$. Therefore $g(2)$ is undefined.

Also $f(4,0) = 4$, $f(4,1) = 3$ and $f(4,2) = 0$. Hence $g(4)$ is defined and $g(4) = 2$.

(b) Clearly $g(n)$ is undefined if n is not a square. Now suppose that $n = m^2$. Then $f(n,y) = m^2 - y^2 > 0$ for $0 \leq y < m$ and $f(n,m) = 0$ so that $g(n) = m$. Thus the domain of g is the set of natural numbers that are squares.

Solution 2.4

We have the following table of values, where $*$ means 'undefined'.

n	0	1	2	3	4	5	6
$g(n)$	$*$	2	$*$	0	3	$*$	1

For example, $g(2)$ is undefined because, although we have $f(2,4) = 0$, $f(2,3)$ is undefined.

Solution 2.5

We have, for all n_1,

$$h(n_1, 0) \approx f(n_1).$$

As $f(n_1) = 1$ for all n_1 this gives

$$h(n_1, 0) = 1, \quad \text{for all } n_1.$$

As $h(n_1, 0)$ is defined for all n_1, we have

$$h(n_1, 1) = h(n_1, 0 + 1) \approx g(n_1, 0, h(n_1, 0)) \approx g(n_1, 0, 1).$$

As $g(n_1, 0, 1)$ is defined and equals $n_1 + 1$ for all n_1, we have

$$h(n_1, 1) = n_1 + 1, \quad \text{for all } n_1.$$

It's only at the next stage that we hit some interesting behaviour! We have

$$h(n_1, 2) = h(n_1, 1 + 1) \approx g(n_1, 1, h(n_1, 1)) \approx g(n_1, 1, n_1 + 1).$$

For $g(n_1, 1, n_1 + 1)$ to be defined, we need $n_1 + 1 \leq 2$, which happens only when n_1 is 0 or 1. This means that for all other values of n_1, namely $n_1 \geq 2$, $h(n_1, 2)$ is undefined, which has the knock-on effect that $h(n_1, 3) \approx g(n_1, 2, h(n_1, 2))$ is undefined and, continuing in this way, $h(n_1, n)$ is undefined for all $n \geq 2$.

Further investigation of the case when $n_1 = 0$ shows that $h(0, 2) = 1$ and consequently that $h(0, n) = 1$, for all $n \geq 2$.

We also find, in the case when $n_1 = 1$, that

$$h(1, 2) = h(1, 1 + 1) \approx g(1, 1, h(1, 1)) \approx g(1, 1, 2),$$

which, as $g(1, 1, 2)$ is defined and equals 3, gives $h(1, 2) = 3$; and that

$$h(1, 3) = h(1, 2 + 1) \approx g(1, 2, h(1, 2)) \approx g(1, 2, 3),$$

which, as $g(1, 2, 3)$ is not defined, means that $h(1, 3)$ and consequently $h(1, n)$ for all $n \geq 3$ are not defined.

To summarize, h can be described as follows:

$$h(n_1, n) = \begin{cases} 1, & \text{for all } n_1 \text{ and } n = 0, \\ n_1 + 1, & \text{for all } n_1 \text{ and } n = 1, \\ 1, & \text{if } n_1 = 0 \text{ and } n \geq 2, \\ 3, & \text{if } n_1 = 1 \text{ and } n = 2, \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

Solution 2.6

One possible solution is to note that

$$h(n) \approx \mu y \, (n = 3y).$$

The relation $n = 3y$ is easily seen to be (primitive) recursive. Thus h is obtained by minimization on a recursive relation and so is a recursive function.

Similar proofs can be used to show that the functions in Examples 2.1, 2.2, 2.3, 2.4 and Problems 2.1, 2.2, 2.3, 2.5 are all recursive.

Solution 2.7

Here $u = 4$ as the program uses the registers R_1, R_2, R_3 and R_4 only. Also $k = 1$. Hence the program given by the proof of Theorem 2.2 which computes $\mu y\,(f(n, y) = 0)$ is as follows.

1	$C(1, 5)$
2	$J(2, 3, 16)$
3	$S(3)$
4	$J(2, 3, 16)$
5	$Z(3)$
6	$J(1, 3, 13)$
7	$J(2, 4, 11)$
8	$S(3)$
9	$S(4)$
10	$J(1, 1, 6)$
11	$Z(4)$
12	$J(1, 1, 6)$
13	$J(2, 4, 15)$
14	$J(1, 1, 16)$
15	$Z(1)$
16	$J(1, 7, 25)$
17	$Z(1)$
18	$Z(2)$
19	$Z(3)$
20	$Z(4)$
21	$S(6)$
22	$C(5, 1)$
23	$C(6, 2)$
24	$J(1, 1, 2)$
25	$C(6, 1)$

Solution 2.8

We count the number of instructions in the program P^* as follows.

Store input values	k
The program P	r
Is $f(n_1, n_2, \ldots, n_k, y) = 0$?	1
Clear registers used by P	u
Add 1 to y	1
Recall input values	k
Put y in R_{k+1}	1
Loop back to P	1
Put y in output register	1

Thus P^* has $2k + r + u + 5$ instructions.

Solution 2.9

Stage	Instruction	R_1	R_2	R_3	Situation number
0	1	4	2	0	$2^1 3^4 5^2 7^0 = 4050$
1	2	4	2	0	$2^2 3^4 5^2 7^0 = 8100$
2	3	4	3	0	$2^3 3^4 5^3 7^0 = 81\,000$
3	4	4	3	1	$2^4 3^4 5^3 7^1 = 1134\,000$
4	1	4	3	1	$2^1 3^4 5^3 7^1 = 141\,750$
5	2	4	3	1	$2^2 3^4 5^3 7^1 = 283\,500$
6	3	4	4	1	$2^3 3^4 5^4 7^1 = 2835\,000$
7	4	4	4	2	$2^4 3^4 5^4 7^2 = 39\,690\,000$
8	1	4	4	2	$2^1 3^4 5^4 7^2 = 4961\,250$
9	5	4	4	2	$2^5 3^4 5^4 7^2 = 79\,380\,000$
10	6	2	4	2	$2^6 3^2 5^4 7^2 = 17\,640\,000$

Solution 2.10

From the calculations we made in Solution 2.9 we see that $S_2(e, 4, 2, 0) = 4050$ and $S_2(e, 4, 2, 9) = 79\,380\,000$. Also the computation halts at stage 10 and hence at all subsequent stages the situation number is the same as at stage 10. Thus

$$S_2(e, 4, 2, 100) = S_2(e, 4, 2, 10) = 17\,640\,000.$$

Solution 2.11

First we must decide if e codes a URM program and, if so, what is this program. We have

$$281\,252 = 281\,250 + 2 = 2^1 3^2 5^6 + 2,$$
$$12 = 6 \times 2,$$
$$18 = 6 \times 3,$$
$$32 = 30 + 2 = 2^1 3^1 5^1 + 2,$$
$$25 = 24 + 1 = 2^3 3^1 + 1.$$

These are the codes of URM instructions, so e is indeed the code number of a URM program. We observe that 281 252 codes the instruction $J(1, 2, 6)$, 12 codes the instruction S(2), 18 codes the instruction S(3), 32 codes the instruction $J(1, 1, 1)$ and 25 codes the instruction $C(3, 1)$. Hence the program coded by e is

1 $J(1, 2, 6)$
2 $S(2)$
3 $S(2)$
4 $S(3)$
5 $J(1, 1, 1)$
6 $C(3, 1)$

With input 4, we have the following trace table and situation numbers.

Stage	Instruction	R_1	R_2	R_3	Situation number
0	1	4	0	0	$2^1 3^4 5^0 7^0 = 162$
1	2	4	0	0	$2^2 3^4 5^0 7^0 = 324$
2	3	4	1	0	$2^3 3^4 5^1 7^0 = 3240$
3	4	4	2	0	$2^4 3^4 5^2 7^0 = 32\,400$
4	5	4	2	1	$2^5 3^4 5^2 7^1 = 453\,600$
5	1	4	2	1	$2^1 3^4 5^2 7^1 = 28\,350$
6	2	4	2	1	$2^2 3^4 5^2 7^1 = 56\,700$
7	3	4	3	1	$2^3 3^4 5^3 7^1 = 567\,000$
8	4	4	4	1	$2^4 3^4 5^4 7^1 = 5670\,000$
9	5	4	4	2	$2^5 3^4 5^4 7^2 = 79\,380\,000$
10	1	4	4	2	$2^1 3^4 5^4 7^2 = 4961\,250$
11	6	4	4	2	$2^6 3^4 5^4 7^2 = 158\,760\,000$
12	7	2	4	2	$2^7 3^2 5^4 7^2 = 35\,280\,000$

The value of $S_1(e, 4, t)$ for $0 \leq t \leq 12$ is given by the situation number at stage t in the table above. The computation halts at stage 12 and hence, for $t > 12$,

$$S_1(e, 4, t) = S_1(e, 4, 12) = 35\,280\,000.$$

Solution 3.1

(a) Let P_f, P_g and P_R be URM programs computing, respectively, the functions f and g and the characteristic function χ_R of the relation R. An informal algorithm for computing h is as follows.

Input $(n_1, n_2, \ldots, n_k)$. Use P_R to compute whether or not $R(n_1, n_2, \ldots, n_k)$ holds. If it holds, use P_f to compute $f(n_1, n_2, \ldots, n_k)$; if it does not hold, use P_g to compute $g(n_1, n_2, \ldots, n_k)$.

(b) The problem is that $h(n_1, n_2, \ldots, n_k)$ is always defined, but

$$f(n_1, n_2, \ldots, n_k)\chi_R(n_1, n_2, \ldots, n_k) + g(n_1, n_2, \ldots, n_k)\,\overline{\mathrm{sg}}(\chi_R(n_1, n_2, \ldots, n_k)) \tag{S.1}$$

might not be. The functions f and g might not be total, in which case there is some $(n_1, n_2, \ldots, n_k)$ for which at least one of $f(n_1, n_2, \ldots, n_k)$ and $g(n_1, n_2, \ldots, n_k)$ is not defined, so that expression (S.1) is not defined. (Expression (S.1) does define a recursive function, but its domain consists of those $(n_1, n_2, \ldots, n_k)$ for which both $f(n_1, n_2, \ldots, n_k)$ and $g(n_1, n_2, \ldots, n_k)$ are defined.)

Solution 3.2

On the assumption that *Tot* is a recursive set, we have found a total recursive function $\Phi : \mathbb{N}^2 \longrightarrow \mathbb{N}$ with the property that if $f : \mathbb{N} \longrightarrow \mathbb{N}$ is a total recursive function then there is some number m such that $\Phi(m, n) = f(n)$ for all $n \in \mathbb{N}$.

Define $g : \mathbb{N} \longrightarrow \mathbb{N}$ by

$$g(n) = \Phi(n, n) + 1.$$

Therefore g is a total recursive function. Hence there is some number m_0 such that

$$g(n) = \Phi(m_0, n)$$

for all $n \in \mathbb{N}$. In particular,

$$g(m_0) = \Phi(m_0, m_0).$$

But, from the definition of g,

$$g(m_0) = \Phi(m_0, m_0) + 1.$$

This is the contradiction that we require.

Solution 3.3

For each $n \in \mathbb{N}$ we shall add n extra instructions to be executed before Q. Each instruction will need to have some sort of neutral effect, which doesn't alter the contents of the registers or cause the new program to halt before Q is executed (although Q might then fail to halt for those inputs for which g is not defined). Our choice is to use Copy instructions from a register to itself. So, for a given $n \in \mathbb{N}$, we take the following concatenated program.

$C(1, 1)$
$C(2, 2)$
$\vdots$
$C(n, n)$
Q

Solution 3.4

(a) The URM program Q with the single instruction

$$Z(1)$$

computes the zero function.

(b) We could mimic the procedure used in Example 3.1. But as the required output of P^* when n is input and $f_P^1(n)$ is defined is independent of n, we can dispense with having to work out the maximum register used by P and copy the input n into an unused register, and simply take P^* to be the following concatenation:

$$P$$
$$Z(1)$$

With input n, this program P^* outputs 0 when $f_P^1(n)$ is defined and fails to halt if $f_P^1(n)$ is not defined. So P^* computes the zero function precisely when the function f_P^1 computed by P is total.

(c) Suppose that the set *Zero* of code numbers of URM programs which compute the zero function is recursive, so that there is an algorithm for testing whether a number m is in *Zero*. This would then give an informal algorithm for testing whether a number e is in *Tot*, as follows. Given e, check whether or not it codes a URM program. If it does not code a program, then e is not in *Tot*. If it does code a program, P say, recover the instructions of P from e and use them to turn P into the corresponding program P^*. Work out the code number $\gamma(P^*)$ of P^* and test whether it is in the set *Zero*. As

$$\gamma(P^*) \in \mathit{Zero} \text{ if and only if } f_P^1 \text{ is total,}$$

we have an algorithm for testing whether e (the code number of P) is in the set *Tot*, so that *Tot* is recursive. This contradicts Theorem 3.2 and we conclude that the set *Zero* is not recursive.

Solution 3.5

(a) The program with code number 8 computes the zero function, which is a total function. Thus if $D(e, 8)$ holds then $W_e = W_8 = \mathbb{N}$, so that the function $f\colon \mathbb{N} \longrightarrow \mathbb{N}$ coded by e is total.

(b) Recall that *Tot* is the set of code numbers of URM programs that compute a total function of one variable. Then

$$e \in \mathit{Tot} \iff e \in \mathit{Prog} \text{ and } D(e, 8).$$

We know from *Unit 2* that *Prog* is a primitive recursive set. Thus, if D were a recursive relation, then *Tot* would be a recursive set. But we know from Theorem 3.2 that this is not the case. Thus D is not a recursive relation.

Unit 2, Theorem 3.10.

SOLUTIONS TO ADDITIONAL EXERCISES

Section 1

1 Consider the program

1 $J(1,2,1)$
2 $Z(1)$
3 $S(1)$

For input 0, instruction 1 is performed repeatedly and the computation does not halt. For positive input, the output is 1. Thus this program computes the function f.

2 It is probably a good idea to investigate f_P^1 first, as the input convention gives the registers very simple initial contents, namely n in register R_1 and 0 in all the other registers. In this case, the effect of instructions 1 to 4 is to add the initial content of register R_1 to that of R_2 and R_3, and then proceed to instruction 5. But as the initial content of R_2 is 0, this means that at instruction 5 the contents of R_1 and R_2 are equal, so that the computation loops on instruction 5 forever and never halts. Thus f_P^1 is a function which is undefined for all inputs n.

Next we consider f_P^2, for which the initial contents of the registers are n_1 in R_1, n_2 in R_2 and 0 in all other registers. The effect of instructions 1 to 4 is still to add the initial content of register R_1 to that of R_2 and R_3, and then proceed to instruction 5. At this stage R_1 contains n_1 and R_2 contains $n_2 + n_1$, which are equal (making the computation loop forever on instruction 5) only when $n_2 = 0$. For all other values the computation halts (on the non-existent instruction 6) with output n_1. Thus f_P^2 is the function

$$f_P^2(n_1, n_2) = \begin{cases} n_1, & \text{if } n_2 > 0, \\ \text{undefined}, & \text{if } n_2 = 0. \end{cases}$$

For f_P^3, suppose that initial contents of registers R_1, R_2, R_3 are n_1, n_2, n_3 respectively. If $n_1 < n_3$, the computation will loop forever round the cycle of instructions $1 \to 2 \to 3 \to 4 \to 1 \to \cdots$ and not halt. If $n_1 \geq n_3$, instructions 1 to 4 will add $n_1 - n_3$ to the initial content of register R_2 and then proceed to instruction 5. If the content of R_2 at this stage, namely $n_2 + n_1 - n_3$, equals that of R_1 (still n_1), the computation doesn't halt; in other cases it halts with output n_1. Thus the function f_P^3 is given by

$$f_P^3(n_1, n_2, n_3) = \begin{cases} n_1, & \text{if } n_1 \geq n_3 \text{ and } n_2 \neq n_3, \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

Section 2

1 We have

$$f(n) \approx \mu z\,(\overline{\text{sg}}(n) + z = 0) + 1.$$

For, if $n > 0$, $\overline{\text{sg}}(n) = 0$ and $\mu z\,(\overline{\text{sg}}(n) + z = 0) = 0$ while, if $n = 0$, $\overline{\text{sg}}(n) = 1$ and $\mu z\,(\overline{\text{sg}}(n) + z = 0)$ is undefined. Thus f is obtained by substitution and minimization from known primitive recursive functions using constants. Hence f is a recursive function.

Alternatively, since f is URM-computable (by Additional Exercise 1 on Section 1), f is recursive (by Theorem 2.6).

2 We have the following table of values for g, where $*$ means 'undefined'.

$g(n,m)$	m 0	1	2	3	4	5
0	0	0	0	$*$	$*$	$*$
1	0	0	0	0	$*$	$*$
2	0	0	0	0	0	$*$
n 3	$*$	0	0	0	0	0
4	$*$	$*$	0	0	0	0
5	$*$	$*$	$*$	0	0	0

We see from this table that

$$f(0) = f(1) = f(2) = 0$$

but that, for $n \geq 3$, $f(n)$ is not defined because, in these cases, $g(n,0)$ is not defined.

3 The given program uses the registers R_1 to R_5 and hence, in the notation of Theorem 2.2, $u = 5$. Also $k = 1$. Hence the program given by Theorem 2.2 which computes $\mu y\,(g(n,y) = 0)$ is as follows.

1 $C(1,6)$
2 $J(2,3,7)$
3 $S(4)$
4 $S(4)$
5 $S(3)$
6 $J(1,1,2)$
7 $C(1,3)$
8 $C(4,2)$
9 $J(1,4,17)$
10 $J(2,3,15)$
11 $S(3)$
12 $S(4)$
13 $S(5)$
14 $J(1,1,9)$
15 $Z(1)$
16 $J(1,1,18)$
17 $C(5,1)$
18 $J(1,8,28)$
19 $Z(1)$
20 $Z(2)$
21 $Z(3)$
22 $Z(4)$
23 $Z(5)$
24 $S(7)$
25 $C(6,1)$
26 $C(7,2)$
27 $J(1,1,2)$
28 $C(7,1)$

4 We give the trace table for the computation using this program with input 1 and the corresponding situation numbers.

Stage	Instruction	R_1	R_2	R_3	Situation number
0	1	1	0	0	$2^1 3^1 5^0 7^0 = 6$
1	2	1	0	0	$2^2 3^1 5^0 7^0 = 12$
2	3	1	1	0	$2^3 3^1 5^1 7^0 = 120$
3	4	1	2	0	$2^4 3^1 5^2 7^0 = 1200$
4	5	1	2	1	$2^5 3^1 5^2 7^1 = 16\,800$
5	1	1	2	1	$2^1 3^1 5^2 7^1 = 1050$
6	6	1	2	1	$2^6 3^1 5^2 7^1 = 33\,600$
7	7	2	2	1	$2^7 3^2 5^2 7^1 = 201\,600$

The situation numbers $S_1(e, 1, t)$ for $0 \leq t \leq 7$ are given in the corresponding rows of this table. The computation halts at stage 7 and hence $S_1(e, 1, t) = S_1(e, 1, 7) = 201\,600$ for $t > 7$.

5 (a) $2 = 2^1 3^0 5^0 7^0$ so the corresponding situation is

Instruction	R_1	R_2	R_3
1	0	0	0

(b) $90 = 2^1 3^2 5^1 7^0$ so the corresponding situation is

Instruction	R_1	R_2	R_3
1	2	1	0

(c) $180 = 2^2 3^2 5^1 7^0$ so the corresponding situation is

Instruction	R_1	R_2	R_3
2	2	1	0

(d) $196 = 2^2 3^0 5^0 7^2$ so the corresponding situation is

Instruction	R_1	R_2	R_3
2	0	0	2

6 The relation that e does not code a URM program is expressed by

$$e \notin \mathit{Prog}$$

where *Prog* is the primitive recursive set of code numbers of URM programs.

The relation that the computation with input n of the URM program with code number e has halted at stage t is expressed by

$$(S_1(e, n, t))_1 > \text{len}(e).$$

Let R be the relation defined by

$$R(e, n, t) \iff (S_1(e, n, t))_1 > \text{len}(e) \text{ or } e \notin \mathit{Prog}.$$

Using a variety of results from *Unit 2*, together with Theorem 2.5 of this unit, we can deduce that R is a primitive recursive relation.

If e does not code a URM program then $R(e, n, t)$ holds for all $t \in \mathbb{N}$ so that $\mu t\, R(e, n, t) = 0$.

If e codes a URM program and the computation with this program halts for input n then $\mu t\, R(e, n, t) = t_0$, where t_0 is the stage at which the computation halts.

If e codes a URM program and the computation with this program does not halt for input n then $\mu t\, R(e, n, t)$ is undefined.

Therefore

$$g(e, n) \approx \mu t\, R(e, n, t).$$

Hence g is obtained from the primitive recursive relation R by minimization, and so g is a recursive function.

7 Suppose that e codes a URM program and the instruction to be carried out at stage t is a Jump instruction.

We know that the code number c of the instruction to be performed at stage t is given by $c = (e)_a$ where $a = (S_k(e, n_1, n_2, \ldots, n_k, t))_1$. Suppose that the instruction is $J(m, n, q)$ so that $c = 2^m 3^n 5^q + 2$. Then we have

$$m = (c \dot{-} 2)_1, \quad n = (c \dot{-} 2)_2, \quad q = (c \dot{-} 2)_3,$$

where the functions $(x, y) \longmapsto (x)_y$ and $\dot{-}$ are primitive recursive.

When the Jump instruction $J(m, n, q)$ is carried out, the numbers in the registers do not change but the number of the next instruction depends on whether the numbers r_m and r_n in the registers R_m and R_n are equal or not. We have

$$r_m = (S_k(e, n_1, n_2, \ldots, n_k, t))_{m+1},$$
$$r_n = (S_k(e, n_1, n_2, \ldots, n_k, t))_{n+1},$$

so the relations '$r_m = r_n$' and '$r_m \neq r_n$' are primitive recursive, and mutually exclusive and exhaustive.

If $r_m = r_n$ then the Jump is carried out and the next instruction number is q. To obtain the situation number at stage $t+1$ we need to change the exponent of 2 in $S_k(e, n_1, n_2, \ldots, n_k, t)$ from a to q. We can achieve this by dividing by 2^a and then multiplying by 2^q. If $r_m \neq r_n$ then all that happens is that the instruction number is increased by 1; so the situation number at stage $t+1$ is obtained by multiplying $S_k(e, n_1, n_2, \ldots, n_k, t)$ by 2. Thus we have

$$S_k(e, n_1, n_2, \ldots, n_k, t+1) = \begin{cases} 2^q \times \text{quot}\,(S_k(e, n_1, n_2, \ldots, n_k, t), 2^a), & \text{if } r_m = r_n, \\ 2 \times S_k(e, n_1, n_2, \ldots, n_k, t), & \text{if } r_m \neq r_n, \end{cases}$$

where the functions mult, exp and quot are primitive recursive. Thus $S_k(e, n_1, n_2, \ldots, n_k, t+1)$ is defined by cases from e and $S_k(e, n_1, n_2, \ldots, n_k, t)$ using known primitive recursive functions and mutually exclusive and exhaustive primitive recursive relations, and so by Theorem 1.5 of *Unit 2* is of the required form.

Section 3

1 We note first that if $f : \mathbb{N} \longrightarrow \mathbb{N}$ is a primitive recursive function, then so also is the function $g : \mathbb{N} \longrightarrow \mathbb{N}$ given by

$$g(n) = \begin{cases} 0, & \text{if } n = 0, \\ \max\{f(k) : 1 \leq k \leq n\}, & \text{if } n \geq 1. \end{cases}$$

The function max used here to define g when $n \geq 1$ is a function $\mathbb{N}^n \longrightarrow \mathbb{N}$. In *Unit 2*, Problem 1.5, we saw that this function is primitive recursive when $n = 2$. Here we need to redefine g so that it is expressed in terms of $\max : \mathbb{N}^2 \longrightarrow \mathbb{N}$.

This is because g has the following definition by primitive recursion

$$g(0) = 0,$$
$$g(n+1) = \max(f(n+1), g(n)),$$

and we have seen in *Unit 2* that the functions max and add are primitive recursive.

Now, let M be the function given by

$$M(n) = \begin{cases} \text{the maximum register number used by } I, & \text{if } n \text{ codes a URM instruction } I, \\ 0, & \text{otherwise.} \end{cases}$$

Then M has the following definition by cases in terms of primitive recursive functions and relations

We know from *Unit 2* that all the component functions and relations of M are primitive recursive.

$$M(n) = \begin{cases} \text{quot}(n+3, 6), & \text{if } n \in \mathit{Zinstr}, \\ \text{quot}(n, 6), & \text{if } n \in \mathit{Sinstr}, \\ \max((n \dot{-} 1)_1, (n \dot{-} 1)_2), & \text{if } n \in \mathit{Cinstr}, \\ \max((n \dot{-} 2)_1, (n \dot{-} 2)_2), & \text{if } n \in \mathit{Jinstr}, \\ 0, & \text{otherwise.} \end{cases}$$

Thus, by Theorem 1.5 of *Unit 2*, M is a primitive recursive function.

The function reg can now be defined by

$$\text{reg}(n) = \begin{cases} \max\{M((n)_k) : 1 \leq k \leq \text{len}(n)\}, & \text{if } n \in \mathit{Prog}, \\ 0, & \text{otherwise.} \end{cases}$$

The function g at the start of this solution is primitive recursive whatever primitive recursive function f is used to define it, so in particular is primitive recursive if the function M is used. Thus, since the function in the first case defining reg is obtained from this version of g and the primitive recursive functions $(n, k) \longmapsto (n)_k$ and len using substitution, it is primitive recursive. Since we also know that *Prog* is a primitive recursive set, we can use Theorem 1.5 of *Unit 2* again to deduce that reg is a primitive recursive function.

2 Since f and g are recursive functions, it follows from Kleene's Normal Form Theorem that there are natural numbers e_1 and e_2 such that, for all $(n_1, n_2, \ldots, n_k) \in \mathbb{N}^k$,

$$f(n_1, n_2, \ldots, n_k) \approx U(\mu z\, T_k(e_1, n_1, n_2, \ldots, n_k, z)),$$

$$g(n_1, n_2, \ldots, n_k) \approx U(\mu z\, T_k(e_2, n_1, n_2, \ldots, n_k, z)),$$

where U and T_k are recursive.

Consider the relations S_1, S_2 and S defined by

$$\begin{aligned} S_1(n_1, n_2, \ldots, n_k, z) &\iff R(n_1, n_2, \ldots, n_k) \text{ and } T_k(e_1, n_1, n_2, \ldots, n_k, z), \\ S_2(n_1, n_2, \ldots, n_k, z) &\iff \text{not } R(n_1, n_2, \ldots, n_k) \text{ and } T_k(e_2, n_1, n_2, \ldots, n_k, z), \\ S(n_1, n_2, \ldots, n_k, z) &\iff S_1(n_1, n_2, \ldots, n_k, z) \text{ or } S_2(n_1, n_2, \ldots, n_k, z). \end{aligned}$$

Using the results for recursive relations equivalent to those for primitive recursive relations in Problem 1.10 of *Unit 2*, we deduce that S_1, S_2 and S are recursive relations.

Now consider $\mu z\, S(n_1, n_2, \ldots, n_k, z)$.

If $R(n_1, n_2, \ldots, n_k)$, then

$$\mu z\, S(n_1, n_2, \ldots, n_k, z) = \mu z\, T_k(e_1, n_1, n_2, \ldots, n_k, z);$$

otherwise

$$\mu z\, S(n_1, n_2, \ldots, n_k, z) = \mu z\, T_k(e_2, n_1, n_2, \ldots, n_k, z).$$

Hence, for all $(n_1, n_2, \ldots, n_k) \in \mathbb{N}^k$,

$$h(n_1, n_2, \ldots, n_k) = U(\mu z\, S(n_1, n_2, \ldots, n_k, z)).$$

Therefore, since U and S are recursive, it follows that h is a recursive function.

3 One example is obtained by letting R be the relation defined by:

$$R(m, n) \iff m \text{ codes a URM program which halts with input } n$$

Then, by Theorem 3.1, the function $f : \mathbb{N}^2 \longrightarrow \mathbb{N}$ given by

$$f(m, n) = \begin{cases} 1, & \text{if } R(m, n) \text{ holds,} \\ 0, & \text{otherwise,} \end{cases}$$

is not recursive.

Now consider the function $g : \mathbb{N}^2 \longrightarrow \mathbb{N}$ given by

$$g(m, n) = \begin{cases} 1, & \text{if } R(m, n) \text{ holds,} \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

Then

$$g(m, n) \approx C_1(\Phi_1(m, n)),$$

where Φ_1 is the universal function for URM-computable functions of one variable. Since Φ_1 is URM-computable, it is recursive, and hence so is g.

4 First we consider the function $f : \mathbb{N}^2 \longrightarrow \mathbb{N}$ defined as follows:

$$f(x,y) = \begin{cases} \beta(J(m,n,y+1)), & \text{if } x \text{ codes the instruction} \\ & J(m,n,q) \text{ with } q > y, \\ x, & \text{otherwise.} \end{cases}$$

Recall that $\beta(I)$ is the code number of the instruction I.

Suppose that x codes the instruction $J(m,n,q)$. Then $x = 2^m 3^n 5^q + 2$ so that

$$m = (x \dot{-} 2)_1, \quad n = (x \dot{-} 2)_2, \quad q = (x \dot{-} 2)_3.$$

Thus we can write the definition of f as

$$f(x,y) = \begin{cases} 2^{(x \dot{-} 2)_1} 3^{(x \dot{-} 2)_2} 5^{y+1} + 2, & \text{if } x \in \mathit{Jinstr} \text{ and } (x \dot{-} 2)_3 > y, \\ x, & \text{otherwise.} \end{cases}$$

Therefore f is a primitive recursive function, since it is defined by cases using primitive recursive functions and relations.

Now, if we have a URM program P which is coded by the number e, the program contains $\text{len}(e)$ instructions. Thus we obtain the standard form of P, that is, the program P', by replacing each instruction of the form $J(m,n,q)$ with $q > \text{len}(e)$ by the instruction $J(m,n,\text{len}(e)+1)$ and leaving the other instructions unchanged. Thus the number which codes the program P' is obtained by replacing each exponent $(e)_r$ for $1 \leq r \leq \text{len}(e)$ by the number $f((e)_r, \text{len}(e))$ where f is the primitive recursive function described above. Hence the number $\gamma(P')$ which codes the program P' is given by the bounded product

$$\prod_{r=1}^{\text{len}(e)} p_r^{f((e)_r, \text{len}(e))}.$$

Thus

$$\text{sf}(e) = \begin{cases} \displaystyle\prod_{r=1}^{\text{len}(e)} p_r^{f((e)_r, \text{len}(e))}, & \text{if } e \in \mathit{Prog}, \\ 0, & \text{otherwise.} \end{cases}$$

Therefore sf is a primitive recursive function, since it is defined by cases using primitive recursive functions and relations.

5 (a) The URM program Q with the single instruction

$$S(1)$$

computes the successor function.

(b) We shall mimic the procedure used in Example 3.1. As the required output of P^* when n is input and $f_P^1(n)$ is defined depends on n, we need to copy the input into an unused register before we apply the program P.

First, as we are given the code number e of a URM program, we recover the instructions of P from e and compute the value of $\text{reg}(e)$ to get the maximum register number $\rho(P)$ used in P. As $\rho(Q) = 1$, the register $\rho(P)+1$ is not used in either P or Q. We then construct P^* as the following concatenation of instructions.

$$\begin{array}{l} C(1, \rho(P)+1) \\ P \\ C(\rho(P)+1, 1) \\ S(1) \end{array}$$

As our program Q only uses register R_1, there is no need to clear the contents of any registers before copying n back to register R_1.

With input n, this program P^* outputs $n+1$ when $f_P^1(n)$ is defined and fails to halt if $f_P^1(n)$ is not defined. So P^* computes the successor function precisely when the function f_P^1 computed by P is total.

6 The zero function is computed by the URM program

1 $Z(1)$

which is coded by the number $2^3 = 8$. Thus

$$n \in \mathit{Zero} \iff n \in \mathit{Prog} \text{ and } E(n, 8).$$

Hence if the relation E were recursive, so too would be the set Zero. Thus we can deduce from the result of Problem 3.4 that the relation E is not recursive.

(This result tells us that the problem of determining whether two URM programs compute the same function of one variable is algorithmically undecidable.)

7 Suppose that the set K is recursive. We show that this leads to a contradiction.

Consider the function h given by

$$h(n) = \begin{cases} \Phi_1(n, n), & \text{if } n \in K, \\ 0, & \text{otherwise.} \end{cases}$$

Note that if $n \in K$ then $\Phi_1(n, n)$ is defined, so that h is total and by Problem 3.1 is also recursive.

It follows immediately that the function $g\colon \mathbb{N} \longrightarrow \mathbb{N}$ given by

$$g(n) = h(n) + 1$$

is also total and recursive. As g is recursive, Theorem 2.9 tells us that there is some natural number e such that, for all n, $g(n) = \Phi_1(e, n)$, that is, the URM program coded by e computes g. Hence

$$g(e) = \Phi_1(e, e).$$

But, as g is total, $e \in W_e$, so that $e \in K$. Therefore, from the definition of g,

$$g(e) = \Phi_1(e, e) + 1,$$

which gives a contradiction.

We conclude that the set K cannot be recursive.

(This result tells us that the problem of determining whether the code of a URM program lies in the domain of the function computed by the program is algorithmically undecidable.)

INDEX

$\approx$ 10
f_P^k 7
$(n)_j$ 20
p_j 5, 20
S_k 21
$\gamma(P)$ 20
$\delta(f)$ 5
μy 9
$\mu y\ (f(n_1, n_2, \ldots, n_k, y) = 0)$ 9, 11
$\mu y\ R(n_1, n_2, \ldots, n_k, y)$ 9
$\rho(P)$ 16
Φ_k 27

reg 33
sf 44
Tot 36

algorithmically computable 29
algorithmically decidable 34
algorithmically solvable 34

Church 29
Church's Thesis 29
Church–Turing Thesis 30
closure under minimization 16, 17
    on a function 16
    on a relation 17
code
    of a primitive recursive function 5
Collatz process 35

decidable, algorithmically 34
decision procedure 34

effectively calculable 29
equality of partial functions 10

formalism, strict 29
function 6
    partial 6
    recursive 14
    total 6
    total recursive 14
    universal 27

general recursive function 30

Halting Problem 35

Kleene 25
Kleene's Normal Form Theorem 26

lambda calculus 31

minimization 9, 11, 16, 17
    closure under 16, 17
    on a relation 9
    partial function defined by 11
    total function defined by 9

normal form 25

partial function 6, 11
    computed by a URM program 7
    defined by minimization 11
    defined by primitive recursion 13
    defined by substitution 13
    URM-computable 7
Post system 31
primitive recursion 13
    partial function defined by 13
program, universal 28

recursive 14
    function 14
    relation 14
    set 14
relation, recursive 14

set, recursive 14
situation number 20
situation of computation 19
stage of computation 19
standard form of URM program 44
strict formalism 29
substitution 13
    partial function defined by 13

total function 6
    defined by minimization 9
total recursive function 14

undefined 6
universal function 27
universal machine 27
universal program 28
URM-computable 7
URM-computable partial function 7